Comparative Analysis of Big Data Stream Processing Systems

Farouk Salem

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 22 June, 2016

Thesis supervisor:

Assoc. Prof. Keijo Heljanko

Thesis advisor:

D.Sc. (Tech.) Khalid Latif

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Farouk Salem

Title: Comparative Analysis of Big Data Stream Processing Systems

Date: 22 June, 2016 Language: English Number of pages: 12+77

Department of Computer Science

Supervisor: Assoc. Prof. Keijo Heljanko

Advisor: D.Sc. (Tech.) Khalid Latif

In recent years, Big Data has become a prominent paradigm in the field of distributed systems. These systems distribute data storage and processing power across a cluster of computers. Such systems need methodologies to store and process Big Data in a distributed manner. There are two models for Big Data processing: batch processing and stream processing. The batch processing model is able to produce accurate results but with large latency. Many systems, such as billing systems, require Big Data to be processed with low latency because of real-time constraints. Therefore, the batch processing model is unable to fulfill the requirements of real-time systems.

The stream processing model tries to address the batch processing limitations by producing results with low latency. Unlike the batch processing model, the stream processing model processes the recent data instead of all the produced data to fulfill the time limitations of real-time systems. The subsequent model divides a stream of records into data windows. Each data window contains a group of records to be processed together. Records can be collected based on the time of arrival, the time of creation, or the user sessions. However, in some systems, processing the recent data depends on the already processed data.

There are many frameworks that try to process Big Data in real time such as Apache Spark, Apache Flink, and Apache Beam. The main purpose of this research is to give a clear and fair comparison among the mentioned frameworks from different perspectives such as the latency, processing guarantees, the accuracy of results, fault tolerance, and the available functionalities of each framework.

Keywords: Big Data, Stream processing frameworks, Real-time analytic, Apache Spark, Apache Flink, Google Cloud Dataflow, Apache Beam, Lambda Architecture

Acknowledgment

I would like to express my sincere gratitude to my thesis supervisor Assoc. Prof. Keijo Heljanko for giving me this research opportunity and providing me continuous motivation, support, and guidance in my research. I also would like to give very special thanks to my instructor D.Sc. (Tech.) Khalid Latif for providing very helpful guidance and directions throughout the process of completing the thesis. I'm glad to thank my fellow researcher Hussnain Ahmed for the thoughtful discussions we had.

Last but not the least, I would like to thank my parent and my wife as the main source of strength in my life and special thanks to my child, Youssof for putting me back in the mood everyday.

Thank you all again.

Otaniemi, 22 June, 2016

Farouk Salem

Abbreviations and Acronyms

ABS	Asynchronous Barrier Snapshot
API	Application Program Interface
D-Stream	Discretized Stream
DAG	Directed Acyclic Graph
FCFS	First-Come-First-Serve
FIFO	First-IN-First-OUT
GB	Giga Byte
HDFS	Hadoop Distributed File System
NA	Not Applicable
MB	Mega Byte
RAM	Random Access Memory
RDD	Resilient Distributed Dataset
RPC	Remote Procedure Call
SLA	Service-Level Agreement
TCP	Transmission Control Protocol
UDF	User-Defined Function
UID	User Identifier
VCPU	Virtual Central Processing Unit
VM	Virtual Machine
WORM	Write Once, Read Many

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are various sensors through which data is gathered by many approaches such as smart devices, satellites, and cameras. These sensors generate a huge amount of data that needs to be stored, processed, and analyzed. There are organizations that collect data for different reasons such as research, market campaigns, different trends detection in stock market, and describing a social phenomenon around the world. These organizations consider the data as the new oil[1]. If data becomes too big, it will be difficult to extract useful information out of it using a regular computer. Such amount of data needs considerable processing power and storage capacity.

1.1 Big Data Processing

There are two approaches to solve the problem of Big Data processing. The first approach is to have a single computer with very high-level computational and storage capacity. The second approach is to connect many regular computers together and building a cluster of computers. In the subsequent approach, storage capacity and processing power are distributed among regular computers. Configuring and managing clusters is a complex process because cluster management has to handle many aspects while dealing with Big Data Systems such as scalability, fault tolerance, robustness, and extensibility.

Due to the high number of components used within Big Data processing systems, it is likely that individual components will fail under any circumstances. This kind of systems face many challenges to deliver robustness and fault tolerance. Therefore, usually data sets should be immutable. Big Data systems should have the already processed data and not just an event state to facilitate recomputing the data when needed. Additionally, component failures can cause some data sets to be unavailable. Thus, data replication on distributed computers is needed [1].

Furthermore, Big Data processing systems should have the ability to process a variety types of data. They should be extensible systems by allowing the integration of new functionalities and supporting other types of data at a minimal cost [2]. Moreover, these systems have to be able to either scale-up or scale-out while data

[1] http://ana.blogs.com/maestros/2006/11/data_is_the_new.html

is growing rapidly [1]. There are two kinds of frameworks which can handle these clusters: Batch processing and Stream processing.

1.1.1 Batch Processing

Batch processing frameworks process Big Data across a cluster of computers. They consider that data has already been collected for a period of time to be processed. Google has innovated a programming model called *MapReduce* for batch processing frameworks. This model takes the responsibility of distributing Big Data jobs on a large number of nodes. Also, it handles node failures, inter-node communications, disk usage and network utilization. Developers need only to write two functions (Map and Reduce) without any consideration of resource and task allocation. This model consists of a file system, a master node, and data nodes [3].

The file system splits the data into chunks, after which it stores the data chunks in a distributed manner on cluster nodes. The master node is the heart of MapReduce model because it keeps track of running jobs and existing data chunks. It has access to meta-data that contains the locations of each data chunk. The data node, called a worker as well, handles storing data chunks and running MapReduce tasks. The worker nodes report to the master node periodically. As a result, the master node becomes able to know which worker nodes are still up, running and ready for performing tasks. If a worker node fails while processing a task, the master node will assign this task to another worker node. Thus, the model becomes fault tolerant [3]. One of the implementations of MapReduce Model is Apache Hadoop[2].

The number of running computers on the cluster and the data-set size are the key parameters that determine the execution time of a batch processing job. It may take minutes, hours or even days [1, 4]. Therefore, batch processing frameworks have high latency to process Big Data. As a result, they are inappropriate to satisfy real-time constraints [5].

1.1.2 Stream Processing

Stream processing is a model for returning results in a low latency fashion. It addresses the high latency problem of the batch processing model. In addition, it includes most of batch processing features such as fault tolerance and resource utilization. Unlike batch processing frameworks, stream processing frameworks target to process data that is collected in a small period of time. Therefore, stream data processing should be synchronized with the data flow. If a real-time system requires X+1 minutes to process data sets which are collected in X minutes, then this system may not keep up with the data volumes and the results may become outdated [6].

Furthermore, storing data in such frameworks is usually based on windows of time, which are unbounded. If the processing speed is slower than the data flow speed, time windows will drop some data when they are full. As a result, the system becomes inaccurate because of missing data [7]. Therefore, all time windows should

[2] http://hadoop.apache.org/ - A framework for the distributed processing of large data sets

have the potential to accommodate variations in the speed of the incoming and outgoing data by being able to slide as and when required [8].

In case of network congestion, real-time systems should be able to deal with delayed, missing, or out-of-order data. They should guarantee predictable and repeatable results. Similar to batch processing systems, real-time systems should produce the same outcome having the same data-set if the operation is re-executed later. Therefore, the outcome should be deterministic [8].

Real-time stream processing systems should be up and available all the time. They should handle node failure because high availability is an important concern for stream processing. Therefore, the system should replicate the information state on multiple computers. Data replication can increase the data volumes rapidly. In addition, Big Data processing systems should store the previous and the recent results because it is common to compare them. Therefore, they should be able to scale-out by distributing processing power and storage capacity across multiple computers to achieve incremental scalability. Moreover, they should deal with scalability automatically and transparently. These systems should be able to scale without any human interaction [8].

1.2 Problem Statement

The batch processing model is able to store and process large data sets. However, this model becomes no longer enough for modern businesses that need to process data at the time of its creation. Many organizations require to process data streams and produce results in low latency to make faster and better decisions in real time. These organizations believe that data is more valuable at the time of generation and data would loss its value over time.

Stream processing systems need to provide the benefits of the batch processing model such as scalability, fault tolerance, processing guarantees, and the accuracy of results. In addition, these systems should process data at the time of its arrival. Due to network congestion, server crashes, and network partitions, some data may be lost or delayed. Consequently, some data might be processed many times, which influences the accuracy of results.

This thesis provides a comparative analysis of selected stream processing frameworks namely Apache Spark, Apache Flink, Lambda Architecture, and Apache Beam. The goal of this research is to provide a clear and fair comparison of the selected frameworks in terms of data processing methodologies, data processing guarantees, and the accuracy of results. In addition, this thesis examines how these frameworks interact with external sources of data such as HDFS and Apache Kafka.

1.3 Thesis Structure

The rest of the thesis is organized as follows. Chapter 2 discusses selected fault tolerance mechanisms which are used in Big Data systems. Chapter 3 explains the architecture of the following stream processing frameworks: Apache Spark, Apache

Flink, Lambda Architecture, and Apache Beam. Chapter 4 provides a comparative analysis of the discussed frameworks from different perspectives such as available functionalities, windowing mechanism, processing and result guarantees, and fault tolerant mechanism. Chapter 5 shows results of experiments on some of the discussed frameworks to emphasis their strengths and weaknesses. The experiments examine the following: the processing time, processing guarantees and the accuracy of results in case of failures of each framework. Furthermore, this chapter discusses the findings of the experiments and possible future extensions. The last chapter provides a summary of what we have achieved.

Chapter 2

Fault Tolerance Mechanisms

Scalability and data distribution have an impact on data consistency and availability. Consistency guarantees that storage nodes hold identical data element for same data chunk at a specific time. Availability guarantees that every request receives a response about whether it succeeded or failed. Availability is affected by component failures such as computer crashes or networking device failures. Networking devices are physical components, which are responsible for interaction and communication between computers on a network[1]. These components may fail, after which the network is split into sub-networks causing network partitioning.

While distributing data among different storage nodes over the network, it is common to have network partitions. As a result, some data may become unavailable for a while. Partition tolerance guarantees that the system continues to operate despite network partitioning[2]. Data replication over multiple storage nodes greatly alleviates this problem [9]. However, if data is replicated in ad-hoc fashion, it can lead to data inconsistency [10].

2.1 CAP Theorem

E. Brewer [11] identifies the fundamental trade-off between consistency and availability in a theorem called the *CAP theorem*. This theorem concerns what happens to data systems when some computers can not communicate with each other within the same cluster. It states that when distributing data over different storage nodes, the data system becomes either consistent or available but not both during the network partitioning. It prohibits perfect both consistency and availability at the same time in the presence of network partitions.

The CAP theorem forces system designers to choose between availability and consistency. The availability focuses on building an available system first and then trying to make it as consistent as possible. This model can lead to inconsistent states between the system nodes which can cause different results on different computers. However, it makes the system available all the time. On the other

[1] https://en.wikipedia.org/wiki/Networking_hardware
[2] https://en.wikipedia.org/wiki/CAP_theorem

hand, the consistency focuses on building a consistent state first and then trying to make it as available as possible. This model can lead to unavailable system when network partitions. It guarantees that a system has identical data when the system is available.

Consistent systems ensure that each operation is executed on all computers. There are many protocols to build a consistent system such as *Two phase commit protocol* [12]. This protocol targets having an agreement from all computers, involved in the operation, before performing that operation. It relies on a coordinator node, which is the master node, to receive client's requests and coordinate with other computers called *cohorts*. The coordinator discards the operation if one of the cohorts fails or does not respond. Figure 2.1 depicts how two phase commit protocol works. Reaching the agreement for the *commit phase* is straightforward if the network and participating computers are reliable. However, this is not always the case in distributed systems due to computer crashes and network partitions.

The two phase commit protocol does not work under arbitrary node failures because the coordinator is a single point of failure. If the coordinator fails, it becomes unable to receive client's requests. Moreover, if the failure happens after the voting phase, cohorts might be blocked because they will be waiting for either a commit or rollback message. Thus, the two phase commit protocol is not fault tolerant.

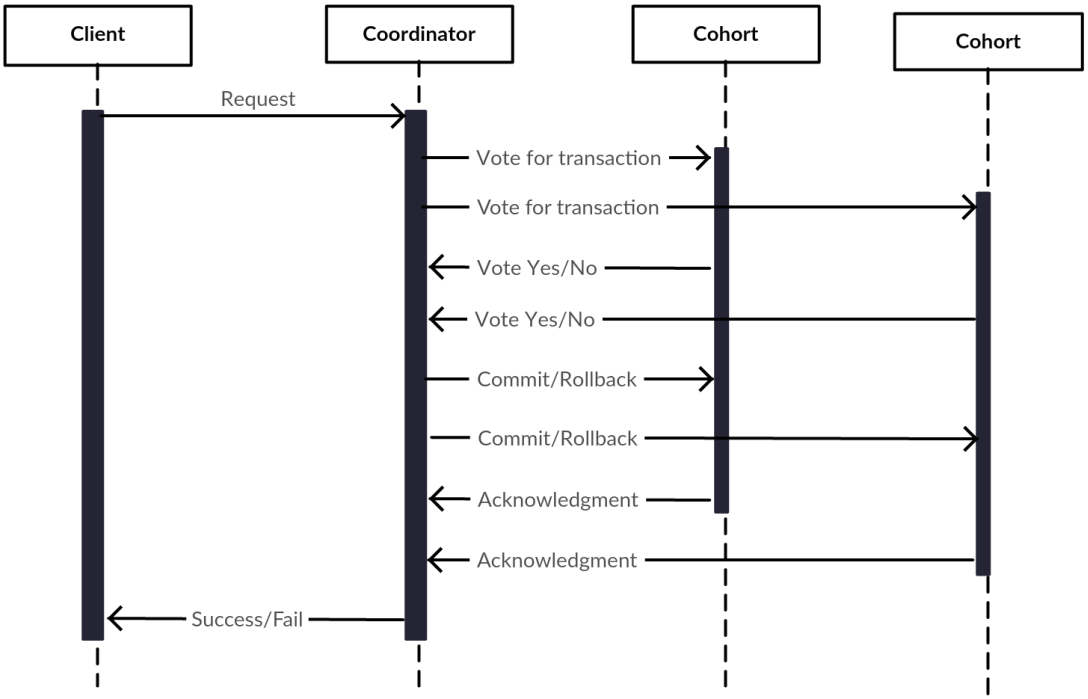

Figure 2.1: Two phase commit protocol

Available systems try to execute each operation on all computers. They keep running if some computers are unavailable. These systems manage network partitions in three steps. Firstly, the system should detect that there is a partition. Secondly, it should deal with this partition by limiting some functionalities, if necessary. Finally, it should have a recovery mechanism when the partition no longer exists. When the partition ends, the recovery mechanism makes all system computers consistent, if possible.

There is not any unified recovery approach that works for all applications while having an available system and network partitions. The recovery approach depends on the available functionalities on each system when the network partitions [11]. For example, Amazon Dynamo [13] is a highly available key-value store. This service targets the following goals: high scalability, high availability, high performance, and low latency. It allows write operations after one replica has been written. Therefore, data may have multiple versions. The system tries to reconcile them. If the system cannot reconcile them, it asks the client for reconciliation in an application specific manner.

2.2 FLP Theorem

In distributed systems, it is common to have failed and slow processes which receive delayed messages. It is impossible to distinguish between them in a fully asynchronous system, which has no upper bound on message delivery. Michael J. Fischer et al. [14] prove that there is no algorithm that will solve the asynchronous consensus problem in a theory called the *FLP theorem*.

Consensus algorithms can have an unbounded run-time, if a slow process is repeatedly considered to be a dead process. In this case, consensus algorithms will not reach a consensus and they will run forever. For example, Figure 2.2 depicts this problem when having a dead or a slow process in a system. Process A wants to reach a consensus for an operation, therefore it sends requests to other processes *B, C, and D*. Process D is slow and Process C is already dead. Although, Process A considers Process D as a dead process as well. As a result, Process A will not reach a consensus. The FLP theorem states that this problem can be tackled if there is an upper bound on transmission delay. Furthermore, an algorithm can be made to solve the asynchronous consensus problem with very high probability to reach a consensus.

2.3 Distributed Consensus

Instead of building a completely available system, a consensus algorithm is used to build a reasonably available and consistent system. Consensus algorithms, such as Paxos [15], Raft [16], and Zab [17], aim at reaching an agreement from the majority of computers. They play a key role in building highly-reliable large-scale distributed systems. These kinds of algorithms can tolerate computer failures. Consensus algorithms help a cluster of computers to work as a group which can survive even when some of them fail.

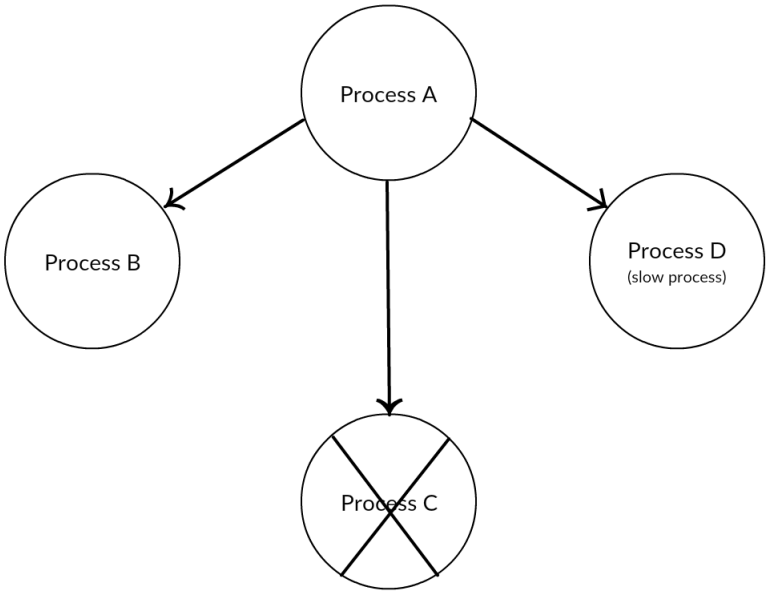

Figure 2.2: Consensus problem

The importance of consensus algorithms arises in the case of replicated state machines [18]. This approach aims to replicate identical copies of a state machine on a set of computers. If a computer fails, the remaining set of computers still have identical state machine. Such a state machine can, for example, represent the state of a distributed database. This approach tolerates computer failures and is used to solve fault tolerance problems in distributed systems. Replicated state machines are implemented using a replicated log. Each log stores the same sequence of operations which should be executed in order. Each state machine processes the same operations in the same order. Consensus algorithms are used to keep the replicated log in a consistent state.

To ensure the consistency through a consensus algorithm, a few safety requirements must be held [19]. If there are many proposed operations from different servers, only one operation is chosen, out of them, at a time. Moreover, each server knows the chosen operation only when it has already been chosen. There are a few assumptions which are considered. Firstly, servers operate at different speeds. Secondly, messages can take different routes to be delivered which affect the communication time. Thirdly, messages can be duplicated or lost but not corrupted. Finally, all servers may fail after an operation has been chosen without other servers knowing it. As a result, these servers will never know about these operations after they are restarted. Consensus algorithms require that a majority of servers are up and able to communicate with each other to guarantee commitment of writes.

2.3.1 Paxos Algorithm

The Paxos algorithm [15] guarantees a consistent result for a consensus if it terminates. Therefore, this algorithm follows the FLP theorem because it may not terminate. It is used as a fault tolerance mechanism. It can tolerate n faults when there are $2n+1$ processes. If the majority, which is $n+1$ processes, agreed on an operation, then this operation can be performed.

The Paxos algorithm has two operator classes: proposers and acceptors. Proposers are a set of processes which propose new operations called *proposals*. Acceptors are a set of processes which can accept the proposals. A single process may act as a proposer and an acceptor. A proposal consists of a tuple of values *{a number, a proposed operation}*. The Paxos algorithm requires the proposal's number to be unique and totally ordered among all proposals. It has two phases to execute an operation: choosing an operation and learning it to others [19]. Figure 2.3 depicts the message flow for a client request in a normal scenario.

2.3.1.1 Choosing an Operation

The Paxos algorithm requires the majority of acceptors to accept a proposal before choosing it. The algorithm wants to guarantee that if there is only one proposal, it will be chosen. Therefore, acceptors must accept the first proposal that they receive. However, different acceptors can accept different proposals. As as result, no single proposal will be accepted by the majority. Acceptors must be allowed to accept more than one proposal but with additional requirement.

Many proposals can be accepted if they have the same operation. If a proposal with a tuple $\{n,v\}$ is chosen, then every proposal with proposal's number greater than n should have the same operation v. All acceptors must guarantee this requirement. Therefore, if a proposal is chosen, every proposal with a higher number than the chosen one must have the same operation. However, if an acceptor fails and then it is restarted, it will not know the highest-numbered chosen proposal. This acceptor must accept the first proposal it receives. The Paxos algorithm transfers the responsibility of knowing the chosen proposal with the highest number from the acceptors side to the proposers side. Each proposer must learn the highest-numbered proposal in order to issue a new proposal. Consequently, the Paxos algorithm guarantees that:

> For each set of acceptors, if a proposal with a tuple $\{n,v\}$ is issued, then it is guaranteed that either no acceptor has accepted any proposal with a number less than n or the operation v is same as the highest-numbered chosen proposal.

The Paxos algorithm follows two phases for choosing an operation. Firstly, a proposer must prepare for issuing a new proposal. It should send a *prepare* request to each acceptor. This request consists of a new proposal with number n. It asks for a promise not to accept any proposal with number less than n. Moreover, it asks for the highest-numbered proposal that has been accepted, if there. When an acceptor receives a *prepare* request with number n that is greater than of any *prepare* requests to which it has already responded, it responds with a promise not to accept any

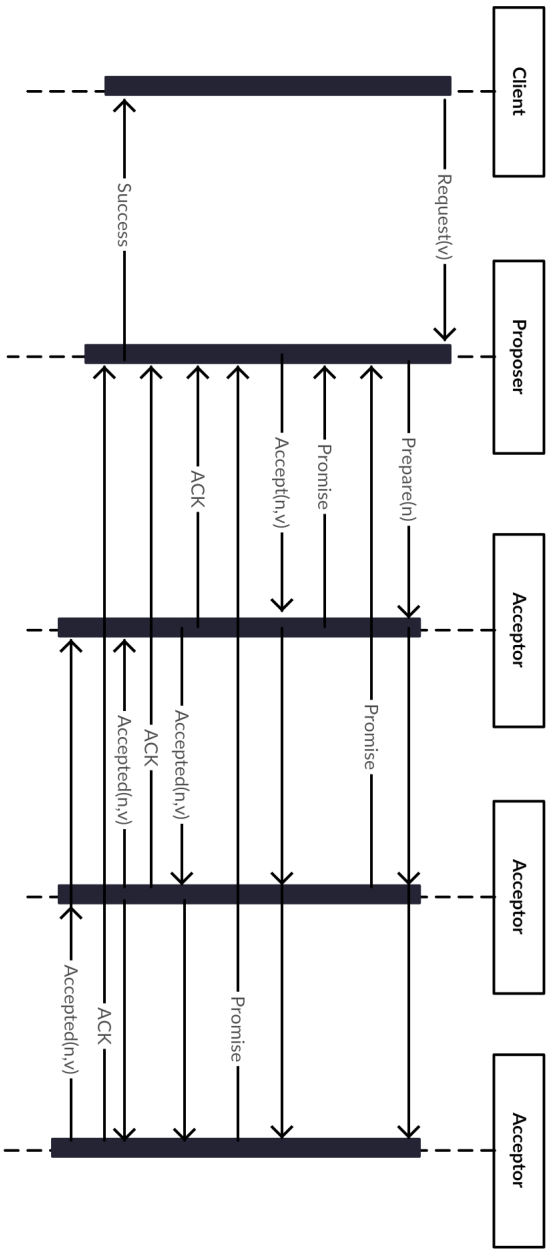

Figure 2.3: Paxos algorithm message flow

proposals with numbers less than n. Additionally, the acceptor may send the highest-numbered proposal that it has accepted.

Secondly, if the proposer receives responses from the majority of acceptors, then it issues a proposal with the number n and an operation v. The operation v is either same as the highest-numbered proposal from one of the acceptors, or any operation if the majority reported that there is no accepted proposals. The proposer sends an *accept* request to each of those acceptors with the issued proposal. If an acceptor receives an *accept* request for a proposal with number n, it accepts the proposal

unless it has already responded to another *prepare* request with greater number than n. This approach guarantees that if a proposal with an operation v is chosen, then every higher-numbered proposal, issued by any proposer, must have the same operation v. However, there is a scenario that prevents accepting any proposal.

Assume that there are two proposers p and q. Proposer p has issued a proposal with number n_1. While proposer q has issued a proposal with number n_2 where $n_2 > n_1$. Proposer p got a promise from the majority of acceptors that they will not accept any proposals with number less than n_1. Then, proposal q got a promise from the majority of acceptors that they will not accept any proposals with number less than n_2. Since $n_1 < n_2$, the acceptors will deny accepting the proposal with number n_1 after their promise to proposer q. This problem causes no progress in issuing and accepting proposals. The Paxos algorithm elects one server to be the only proposer in the system at a time to solve this problem. This proposer will send *prepare* and *accept* requests to acceptors. However, this approach can affect the liveness of the system if only one proposer has a lock forever. This problem can be tackled by setting a timeout for each lock, which the lock is released.

2.3.1.2 Learning a Chosen Operation

After issuing proposals and choosing an operation, acceptors must know that a proposal has been accepted by the majority to be performed. When an acceptor accepts a proposal, it informs other acceptors. When an acceptor receives an accepted proposal from the majority of acceptors, it becomes able to perform the operation of the chosen proposal.

Due to message loss and server crashes, a proposal could be chosen and an acceptor will never know about it. Acceptors can ask each other about the accepted proposals. However, failures of multiple acceptors at the same time can prevent an acceptor to know whether a proposal has been accepted by the majority. In this case, the acceptor will never know about the chosen proposal unless a new one has been issued. As a result, acceptors will inform each other about the accepted proposals. Thereby, this acceptor will find out the chosen operation by the majority.

One of the Paxos algorithm usages is to perform operations on a cluster of computers. All computers should perform the operations in the same order. As a result, their state machines become identical. The Paxos algorithm is used for implementing identical state machines on distributed computers.

2.3.1.3 Implementing a State Machine

The straightforward way to implement a distributed system is by sending all requests to a central server, after which this server broadcasts them to other servers in a specific order. This central server acts as a proposer to guarantee better progress of proposals. However, this server becomes a single point of failure. Therefore, the Paxos algorithm considers that each server is able to act as proposer and acceptor. However, practical Paxos implementations select a single proposer with a leader election algorithm.

The leader becomes the only proposer in the system. It receives requests and organizes them in order. The Paxos algorithm guarantees the order of requests by

proposal's numbers. For each request, the leader issues a new proposal with a number greater than the issued proposals. Then, it sends *prepare* and *accept* requests to other servers. When the leader fails, a new leader is elected. The new leader may have some missing proposals. The leader tries to fill them by sending a *prepare* request for each missing proposal. The leader cannot consider new requests before filling the gaps. If there is a gap that should be filled immediately, the leader sends a special request that leaves the state machine unchanged.

2.3.2 Raft Algorithm

The Paxos algorithm has proven its efficiency, safety, and liveness. Nonetheless, it has shown to be fairly difficult to understand [16]. Also, it does not have a unified approach for implementation. The Raft algorithm [16] is similar to the Paxos algorithm but is easier to understand by having a more straightforward approach.

The Raft algorithm has three different states for servers: *leader*, *follower*, and *candidate*. Only one server can act as a leader at a time, and other servers are followers. The leader receives client's requests and sends them to the followers. Followers respond to the leader's requests without issuing any of them. If a follower receives a client's request, it passively redirects the request to the leader. The leadership lasts for a period of time called *term*. Figure 2.4 presents the cases how servers are changing their states.

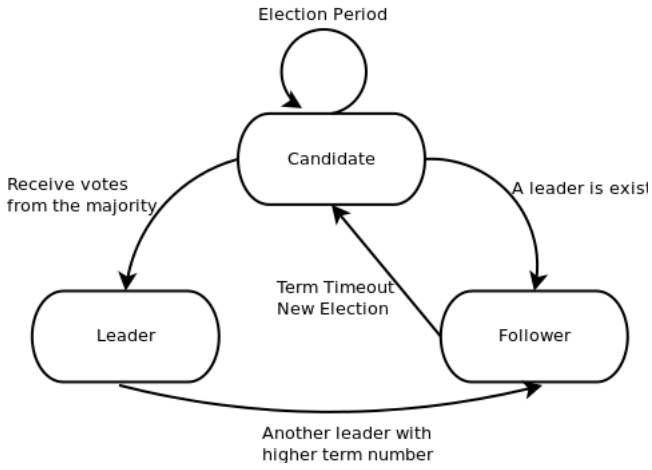

Figure 2.4: Raft state diagram

The Raft algorithm divides the running time into terms which are numbered in an ascending order. The duration of the terms is random, after which terms will expire. Each term has a leader which receives all requests, and stores them in a local log. Then, it sends them to the followers. If a server discovers that it has a smaller term number, the server updates its term number. On the other hand, if a server discovers that it has a higher term number, it rejects the request. If a leader or a candidate discovers that its term number is out-of-date, it converts itself to a follower state. Term numbers are exchanged between servers while communicating.

Communication between Raft cluster is based on RPCs. There are two types of RPCs: *RequestVote* and *AppendEntries*. *RequestVote* RPCs are used at the beginning of a term for electing a new leader. *AppendEntries* RPCs are used to send requests to followers. Figure 2.5 describes how the Raft algorithm works in normal scenario. Leader's requests are new logs to be appended or heartbeats to keep the leadership.

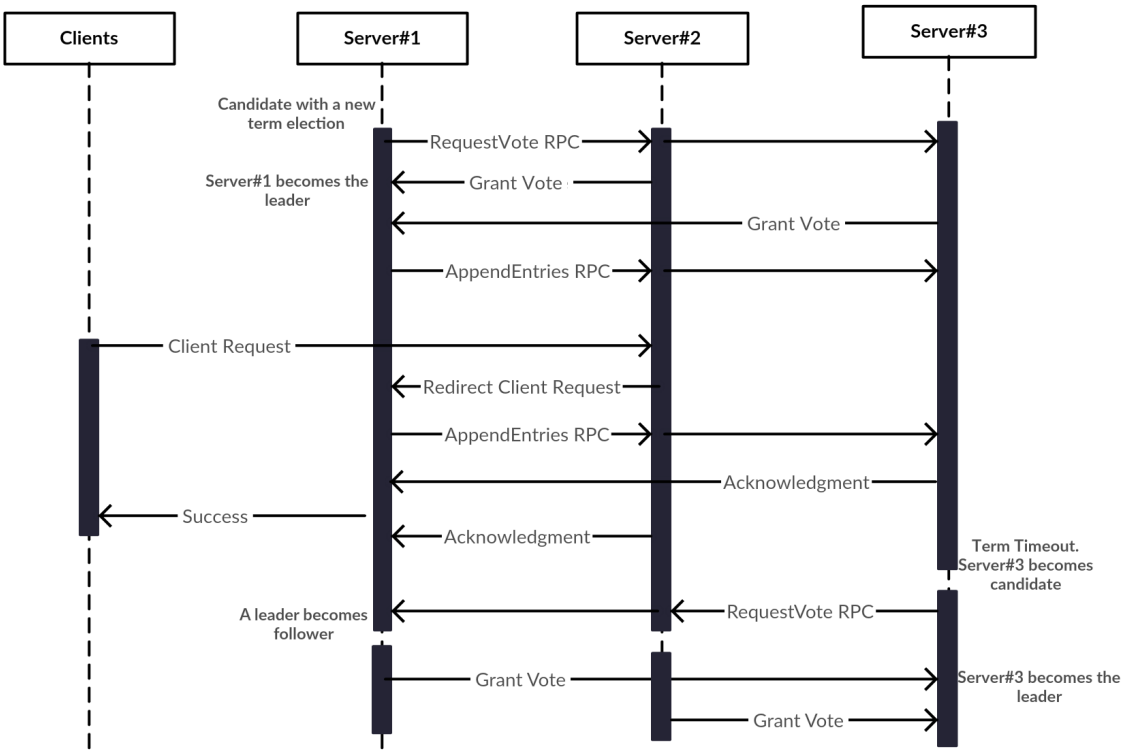

Figure 2.5: Raft sequence diagram

2.3.2.1 Leader election

When there is no leader, all servers go to the candidate state for electing a new leader with a new term number. If a candidate wins, it acts as a leader for rest of the term. If electing a leader fails, a new term will start with a new leader election. When a leader is elected and as long as followers are receiving valid RPCs, they remain in the follower state.

When a follower does not receive any RPCs for a specific period of time called *election timeout*, the follower changes its state to a candidate state. It increases the current term number for a new leader election. Then, it issues *RequestVote* RPCs to all servers for voting itself as a new term leader. A candidate has three scenarios which can happen after voting for a new leadership.

Firstly, a candidate wins an election if it receives acceptance votes from the majority of servers. Each server votes at most once in each term. Servers follow First-Come-First-Serve(FCFS) approach while accepting votes. Choosing a leader by the majority of servers ensures that at most one candidate will be elected as a leader.

Once a candidate is elected to be the term leader, it receives client requests and sends *AppendEntries* RPCs to other servers.

Secondly, while a candidate is waiting for votes, it may receive an *AppendEntries* RPC from another leader term. If the RPC's term number is larger than the candidate's term number, then the candidate converts its state to a follower state and updates its current term to the leader's term number. If the candidate receives the RPC with a smaller term number, the candidate rejects it and continues in the candidate state.

Finally, if a candidate neither wins nor loses, it will start a new vote after the election timeout. This can happen because many followers become candidates at the same time. As a result, each one votes to be a leader at the same time with no progress. The Raft algorithm uses randomized election timeout to ensure that having such scenario is rare. The random election timeout for each candidate ensures that this problem will never happen in two terms respectively.

Raft has a restriction in electing a new leader. The new leader should be up-to-date with the majority of candidates. The voting process prevents electing a leader that has less log entries. When a candidate votes itself, it sends *RequestVote* RPCs to all candidates. Each *RequestVote* RPC contains the following fields:

- Term: The candidate's new term number, which is the last term + 1.
- Candidate Id: The id of the candidate which is requesting vote.
- Last Log Index: The candidate sends its last log index, so that other candidates can check whether this candidate has at least what they have in their logs.
- Last Log Term: This is the term number for the last log index. This helps other candidates to check whether the last index belongs to a correct term number according to their logs.

The receiver candidate grants vote if its log is at least up-to-date to the sender candidate's log. If the receiver candidate has more entries than the sender candidate or it has already voted to another candidate, it replies through an RPC with a rejection response.

2.3.2.2 Log Replication

Once a leader is elected, it manages client's requests. Each client request contains an operation to be committed in the state machine. Once the leader receives a request, it stores the request's operation in the local log as a new entry. Then, it creates *AppendEntries* RPCs in parallel for each follower in the Raft cluster. When the majority of followers adds the RPCs to their logs, the leader executes the operation to its state machine and returns the result to the client. The executed operation is called *committed*. The leader tells followers about the committed entries when sending *AppendEntries* RPCs. If some followers do not acknowledge an RPC message due to slow process, message loss, or server crashes, the leader retries sending the

RPC until all servers become eventually synchronized. The Raft algorithm follows a specific methodology to ensure consistency between all servers.

Each log entry consists of a term number, an operation, and an index. The index must be identical on all servers. The leader sorts and manages the indexes of the log entries. Moreover, it keeps track of the last committed log entry. It distributes the log entries via RPCs. Each *AppendEntries* RPC consists of the following:

- Term: This is the current term number. Followers use it to detect fake leaders and make sure of consistency.
- Leader Id: Followers use this id to redirect client's requests to the leader, if there.
- Previous Log Index: Followers use this index to make sure that there are not any missing entries and they are consistent with the leader.
- Previous Log Term: Followers use it to make sure that the previous log index belongs to the same term number. This is useful when a new leader is elected. Followers can ensure consistency.
- Array of entries: This array consists of a list of ordered entries which followers should add them to their logs. In case of HeartBeat RPCs, this field is empty.
- Leader Commit: This field contains the last committed log index. Followers use this field to find out which operations should be executed to their state machines.

Followers reply to the *AppendEntries* RPCs by another RPC to inform the leader whether the log entries are added. This RPC reply is essential in case of leader failure.

A leader can fail after adding entries to its log without replicating them. Also, it can fail after committing a set of operations to its state machine without informing others. When a new leader is elected, it handles inconsistencies. The leader forces the followers to follow its log; the leader never overrides its log. When the leader sends *AppendEntries* RPCs to followers with a specific *Previous Log Index*, it waits for a result RPC from each follower. If the leader receives a failure RPC from a follower, it recognizes that they are inconsistent. Therefore, it sends another *AppendEntries* RPC to this follower but with less *Previous Log Index*. The leader keeps decreasing this index until the follower eventually succeeded. The follower removes all log entries after the succeeded index. The leader sends all log entries after that index. Therefore, they become in a consistent state.

2.3.3 Zab Algorithm

The original Paxos algorithm requires that if there is an outstanding operation, all issued proposals should have the same operation. Therefore, it does not enable multiple outstanding operations. The Zab algorithm [17] is used as an atomic broadcast algorithm. Unlike the Paxos algorithm, it guarantees total delivery operations

order based on First-In-First-Out (FIFO) approach. It assumes that each operation depends on the previous one. It gives each operation a *state change number* that is incremented with respect to the state number of the previous operation. The Zab algorithm assumes that each operation is idempotent; executing the same operation multiple times does not lead to an inconsistent state. Therefore, it guarantees *exactly once* semantics.

The Zab algorithm elects a leader to order client's requests. Figure 2.6 describes the phases of Zab protocol. After the leader election phase, a new leader starts a new phase to discover the highest *identification schema*. Then, the leader synchronizes its log with the discovered highest schema. After being up-to-date, the leader starts to broadcast its log to other servers. The Zab algorithm follows a different mechanism than the Raft algorithm in case of recovering leader failures.

Figure 2.6: Zab protocol[3]

The recovery mechanism is based on *identification schema* that enables the new leader to determine the correct sequence of operations to recover state machines. Unlike the Raft algorithm, the new leader updates its log entries. Additionally, it is not mandatory for the new leader to be up-to-date. Each operation is identified by the *identification schema* and the position of this operation in its schema. Only a server with the highest identification schema can send the accepted operations to the new leader. Thereby, only the new leader is responsible for deciding which server is up-to-date using the highest schema identifier.

Communication between different servers is based on bidirectional channels. The Zab algorithm uses TCP connections to satisfy its requirements about integrity and delivery order. Once the new leader recovers the latest operations, it starts to establish connections with other servers. Each connection is established for a period of time called *iteration*. The Zab algorithm is used in Apache Zookeeper[4] to implement a primary-backup schema.

2.4 Apache ZooKeeper

Apache ZooKeeper [20] is a high-performance coordination service for distributed systems. It enables different services such as group messaging and distributed locks in a replicated and centralized fashion. It aims to provide a simple and high performance coordination kernel for building complex primitives.

[3] http://www.tcs.hut.fi/Studies/T-79.5001/reports/2012-deSouzaMedeiros.pdf
[4] https://zookeeper.apache.org/ - A coordination distributed service

Instead of implementing locks on primitives, ZooKeeper implements its services based on *wait-free* data objects which are organized in hierarchy similar to file systems. It guarantees FIFO ordering for client operations. It implements the ordering mechanism by following a simple pipelined architecture that allows buffering many requests. The wait-free property and FIFO ordering enable efficient implementation for performance and fault tolerance.

ZooKeeper uses replication to achieve high availability through comprising of a large number of processes to manage all coordination aspects. It uses the Zab algorithm to implement a leader-based broadcast protocol for coordinating between different processes. The Zab algorithm is used in update operations but it is not used in read operations because ZooKeeper servers manage them locally. It guarantees consistent hierarchical namespace among different servers. However, ZooKeeper implements the Zab protocol in a different way[3]. Figure 2.7 shows the phases that are implemented. ZooKeeper combines Phases 0 and 1 of the original Zab protocol in a phase called *Fast Leader Election*. This phase attempts to elect a leader that has the most up-to-date operations in its log. Thereby, the new leader does not need the *Discovery* phase to find the highest *identification schema*. The new leader needs to recover the operations to broadcast. These operations are organized in hierarchical namespaces.

Figure 2.7: Zab protocol in ZooKeeper[3]

ZooKeeper manages data nodes according to hierarchical namespaces called *data trees*. There are two types of data nodes, which are called *znodes*:

- Regular znodes: Clients manage to create and delete them explicitly,
- Ephemeral znodes: Clients create this type of znodes. Clients can either delete them explicitly or the system removes them when the client's session is terminated.

Data trees are based on UNIX file system paths. Figure 2.8 depicts the hierarchical namespace in ZooKeeper. When creating a new znode, a client sets a monotonically increasing sequential value. This value must be greater than the value of the parent znode. Znodes under the same parent znode must have unique values. As a result, ZooKeeper guarantees that each znode has a unique path which clients can use to read data.

When a client connects to ZooKeeper, the client initiates a session. Each session has a timeout, after which ZooKeeper terminates the session. If the client does not send any requests for more than the timeout period, the ZooKeeper service

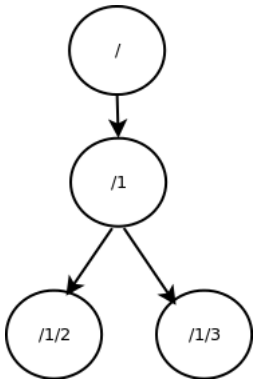

Figure 2.8: ZooKeeper hierarchical namespace

considers it as a faulty client. While terminating a session, ZooKeeper deletes all ephemeral znodes related to this client. During the session, clients use an API to access ZooKeeper services.

Client API: ZooKeeper exposes a simple set of services in a simple interface which clients can use for several services such as configuration management and synchronization. The client API enables clients to create and delete data within a specific path. Moreover, clients can check whether a specific data exists, in addition to changing and retrieving this data.

Clients can set *a watch flag* when requesting a read operation on a znode to be notified if the retrieved data has been changed. ZooKeeper notifies clients about changes without the need of polling the data itself again. However, if data is big, then managing flags and data change notifications may consume a reasonable amount of resources and need larger latency. Therefore, ZooKeeper is not designed for data storage. It is designed for storing meta-data information or configuration about the data itself. Clients can access only data trees which they are allowed to.

ZooKeeper manages access rights with each sub-tree of the data tree using a hierarchical namespace. Clients have the ability to control who can access their data. Additionally, ZooKeeper implements the client API in both synchronous and asynchronous fashion. Clients use the synchronous services when they need to perform one operation to ZooKeeper at a time. While they use the asynchronous services when they need to perform a set of operations in parallel. ZooKeeper guarantees that it responds to the corresponding callbacks for each service in order.

ZooKeeper guarantees a sequential consistency, timeliness, atomicity, reliability, and single image view. The sequential consistency is guaranteed by applying the operations in the same order which they were received by the leader. Atomicity of transaction is guaranteed by each action either succeeding or failing. ZooKeeper can be configured to guarantee that any server has the same data view. In addition, it guarantees the reliability of updates by making sure that once an update operation is persisted, the update effect remains until overriding it. One of the systems which uses ZooKeeper is Apache Kafka[5].

[5] http://kafka.apache.org/ - A high-throughput distributed messaging system

2.5 Apache Kafka

Apache Kafka [21] is a publisher-subscriber, distributed, scalable, partitioned and replicated log service. It offers a high throughput messaging service. It aims to store data in real time by eliminating the overhead of having rich delivery guarantees. Kafka consists of a cluster of servers, each of which is called *broker*. Brokers maintain feeds of messages in categories called *topics*. A topic is divided into *partitions* for load balancing. Partitions are distributed among brokers. Each partition contains a sequence of messages. Each message is assigned a sequential id number called *offset*, which is unique within its partition. Clients add messages to a topic through *producers* and they read the produced messages through *consumers*.

For each topic, there is a pool of producers and consumers. Producers publish messages to a specific topic. Consumers are organized in *consumer groups*. Consumer groups are independent and no coordination is required among them. Each consumer group subscribes to a set of topics and creates one or more message streams. Each message stream iterates over a stream of messages. Message streams never terminate. If there is no more messages to be consumed, the message stream's iterator blocks consuming until new messages are produced. Kafka offers two consumption delivery models: point-to-point and publish/subscribe. In point-to-point model, a consumer group consumes a single copy of all messages in a topic. In this model, Kafka delivers each message to only one consumer for each consumer group. In publish/subscribe model, each consumer consumes all the message in a topic. Consumers can only read messages sequentially. This allows Kafka to deliver messages in high throughput through a simple storage layer.

The simple storage layout helps Kafka to increase its efficiency. Kafka stores each partition in a list of logical log files, each of which has the same size. When a producer produces a message to a topic, the broker appends this message to the end of the current log file of the corresponding partition. Kafka flushes files after a configurable number of time for better performance. Consumers can only read the flushed messages. If a consumer acknowledges that it has received a specific offset, this acknowledgement implies that it has received all messages prior to that offset. Kafka does not support random access because messages do not have explicit message ids but instead have logical offsets. Brokers manage partitions in a distributed manner.

Each partition has a leader broker which handles all read and write requests to that partition. Partitions are designed to be the smallest unit of parallelism. Each partition is replicated across a configurable number of brokers for fault tolerance. Partitions are consumed by one consumer within each consumer group. This approach eliminates the needs of coordination between different consumers within the same consumer group to know whether a specific message is consumed. Moreover, it eliminates the needs of managing locks and state maintenance overhead which affect the overall performance. Consumers within the same consumer group need only to coordinate when re-balancing the load of consuming messages.

Consumers read messages by issuing asynchronous poll requests for a specific topic and partition. Each poll request contains a beginning offset for the message consumption in addition to number of bytes to fetch. Consumers receive a set of

messages from each pull request. Kafka implements *a consumer API* for managing requests. The consumer API iterates over the pulled messages to deliver one message at a time. Consumers are responsible for specifying which offsets are consumed. Kafka gives consumers more flexibility to read same offsets more than once. It designs brokers to be stateless.

Brokers do not maintain the consumed offsets for each consumer. Therefore, they become blind of the consumed messages to be deleted. Kafka has a time-based SLA for message retention. After a configurable time of producing a message, Kafka automatically deletes it. However, Kafka enables storing data forever. This approach reduces the complexity of designing brokers and utilizes the performance of message consumption. Additionally, Kafka utilizes the network consumption while producing messages. It allows producers to submit a set of messages in a single send request. Furthermore, It utilizes brokers' memory by avoiding caching messages in memory. Instead, it uses the file-system page cache. When a broker restarts, it retains the cache smoothly. This approach makes Kafka avoiding the overhead of the memory garbage collection. Moreover, Kafka avoids having a central master node for coordination. It is designed in a decentralized fashion.
Distributed Coordination: Figure 2.9 describes Kafka components and how they use Apache ZooKeeper. Apache Kafka uses ZooKeeper to coordinate between different system components as following:

- ZooKeeper detects the addition and the removal of brokers and consumers. When a new broker joins Kafka's cluster, the broker stores its meta-data information in *a broker registry* into ZooKeeper. The broker registry contains its host name, port, and partitions which it stores. Similarly, when a new consumer starts, it stores its meta-data information in *a consumer registry* into ZooKeeper. The consumer registry consists of the consumer group, which this consumer belongs to, and the list of subscribed topics.

- ZooKeeper triggers the re-balance process. Consumers maintain watch flags in ZooKeeper for the subscribed topics. When there is a change in consumers or brokers, ZooKeeper notifies consumers about this change. Consumers can initiate a re-balance process to determine the new partitions to consume data from. When a new consumer or a new broker starts, ZooKeeper re-balances the consumption over brokers to utilize the usage of all brokers.

- ZooKeeper maintains the topic consumption. It associates an *ownership registry* and an *offset registry* for each consumer group. The *ownership registry* stores which consumer consumes which partition. The *offset registry* stores the last consumed offset for each partition. When a new consumer group starts and no *offset registry* exists, consumers start consuming messages from either smallest or largest offset. When a consumer starts consuming from a specific partition, the consumer registers itself in the *ownership registry* as the owner of this partition. Consumers periodically update the *offset registry* with the last consumed offsets.

Kafka uses ephemeral znodes to create broker registries, consumer registries, and ownership registries. While it uses regular znodes to create offset registries. When a broker fails, ZooKeeper automatically removes its broker registry because this registry is no longer needed. Similarity, when a consumer fails, it removes its consumer registry. However, Kafka keeps the offset registry to prevent the consumption of the consumed offsets when a new consumer joins the same consumer group.

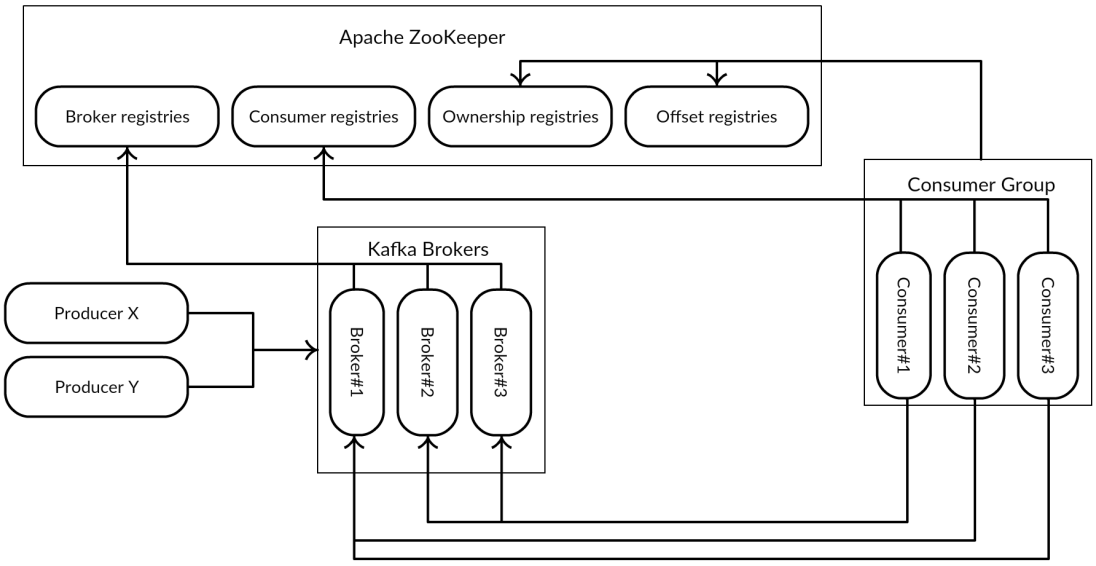

Figure 2.9: Apache Kafka architecture

Chapter 3

Stream Processing Frameworks

Batch processing frameworks need a considerable amount of time to process Big Data and produce results. Modern businesses require producing results in low latency. The demand for real-time access to information is increasing. Many frameworks have been proposed to process Big Data in real time. However, they face challenges, as discussed in Section 1.1.2 in processing data in low latency and producing results with high throughput. In addition, they face challenges in producing accurate results and dealing with fault tolerance. Unlike batch processing frameworks, stream processing frameworks collect records into data windows [22].

Data windows divide data-sets into finite chunks for processing as a group. Data windowing is required to deal with unbounded data streams. While it is optional to process bounded data-sets. A data window is either aligned or unaligned. Aligned window is applied across all the data for a window of time. Unaligned window is applied across a subset of the data from a window of time according to specific constraints such as windowing by key.

There are three types of windows when dealing with unbounded data streams[1]: fixed, sliding, and sessions. Fixed windows are defined by a fixed window size. They are aligned in most cases, however they are sometimes unaligned by shifting the windows for each key. Fixed windows do not overlap. Sliding windows are defined by a window size and a sliding period. They may overlap if the sliding period is less than the window size. A sliding window becomes a fixed window if the sliding period equals to the window size. Sessions capture some activities over a subset of data. If a session does not receive new records related to its activities for a period of time called *a timeout*, the session is closed.

Stream processing frameworks can process data according to its time domains such as event time and processing time. Event time domain is the time at which a record occurs or is generated. Processing time domain is the time at which a record is received within a processing pipeline. The best case for a record is when its event time equals to its processing time, but this is not the case in distributed systems. Apache Spark, Apache Flink, and Apache Beam are some of the frameworks that deal with Big Data stream processing.

[1] https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101

3.1 Apache Spark

Apache Spark is an extension to the MapReduce model that efficiently uses cluster's memory for data sharing and processing among different nodes. It adds primitives and operators over MapReduce to make it more general. It supports batch, streaming, and iterative data processing. Spark uses a memory abstraction model called *Resilient Distributed Dataset* [23] that efficiently manages cluster's memory. Moreover, it implements *Discretized Streams* [24] for stream processing on large clusters.

3.1.1 Resilient Distributed Datasets

The MapReduce model lacks in dealing with applications which iteratively apply a similar function on the same data at each processing step such as machine learning and graph processing applications. These kinds of applications need iterative processing model. The MapReduce model deals badly with iterative applications because it stores data on hard drives and reloads the same data directly from hard drives on each step. MapReduce jobs are based on replicating files on distributed file system for fault recovery. This mechanism adds a significant overhead on network transmission when replicating large files. *Resilient Distributed Dataset (RDD)* is a fault-tolerant distributed memory abstraction that avoids data replication and minimizes disk seeks. It allows applications to cache data in memory across different processing steps which leads to substantial speedup on future data reuse. Moreover, RDDs can remember the operations used to build them. When a failure happens, they can re-build the data with minimal network overhead.

RDDs provide some restrictions on the shared memory usage for enabling low-overhead fault tolerance. They are designed to be read-only and partitioned. Each partition contains records which can be created through deterministic operations called *transformations*. Transformations include map, filter, groupBy, and join operations. RDDs can be created through other existing RDDs or a data set in a stable storage. These restrictions facilitate reconstructing lost partitions because each RDD has enough information about how to rebuild it from other RDDs. When there is a request for caching an RDD, Spark processing engine stores its partitions by default in memory. However, partitions may be stored on hard drives when enough memory space is not available or users ask for caching on hard drives only. Furthermore, Spark stores big RDDs on hard drives because they can never be cached to memory. RDDs may be partitioned based on a key associated with each record, after which they are distributed accordingly.

Each RDD consists of a set of partitions, a set of dependencies on parent RDDs, a computing function, and meta-data about its partitioning schema. The computing function defines how an RDD is built from its parent RDDs. For example, if an RDD represents an HDFS file, then each partition represents a block of the HDFS file. The meta-data will contain information about the HDFS file such as location and number of blocks. This RDD is created from a stable storage, therefore it does not have dependencies on parent RDDs. A transformation can be performed on some RDDs, which represent different HDFS files. The output of this transformation is

a new RDD. This RDD will consist of a set of partitions according to the nature of the transformation. Each partition will depend on a set of other partitions from the parent RDDs. Those parent partitions are used to recompute the child partition when needed.

Spark divides the dependencies on parent RDDs into two types: narrow and wide. Narrow dependencies indicate that each partition of a child RDD depends on a fixed number of partitions in parent RDDs such as *map* transformation. Wide dependencies indicate that each partition of a child RDD can depend on all partitions of the parent RDDs such as *groupByKey* transformation. This categorization helps Spark to enhance the execution and recovery mechanism. RDDs with narrow dependencies can be computed within one cluster node such as *map* operation followed by *filter* operation. In contrast, wide dependencies require data from parent partitions to be shuffled across different nodes. Furthermore, a node failure can be recovered more efficiently with a narrow dependency because Spark will recompute a fixed number of lost partitions. These partitions can be recomputed in parallel on different nodes. In contrast, a single node failure might require a complete re-execution in case of wide dependencies. In addition, this categorization helps Spark to schedule the execution plan and take checkpoints.

Spark job scheduler uses the structure of each RDD to optimize the execution plan. It targets to build a Directed Acyclic Graph (DAG) of computation stages for each RDD. Each stage contains as many transformations as possible with narrow dependencies. A new stage starts when there is a transformation with wide dependencies or a cached partition that is stored on other nodes. The scheduler places tasks based on data locality to minimize the network communication. If a task needs to process a cached partition, the scheduler sends this task to a node that has the cached partition.

3.1.2 Discretized Streams

Spark aims to get the benefits of the batch processing model to build a streaming engine. It implements *Discretized Streams (D-streams)* model to deal with streaming computations. D-Streams model breaks down a stream of data into a series of small batches at small time intervals called *Micro-batches*. Each micro-batch stores its data into RDDs. Then, the micro-batch is processed and its results are stored in intermediate RDDs.

D-Streams provide two types of operators to build streaming programs: *output* and *transformation*. The *output* operators allow programs to store data on external systems such as HDFS and Apache Kafka. There are two output operators: *save* and *foreach*. The *save* operator writes each RDD in a D-stream to a storage system. The *foreach* operator runs a User-Defined-Function (UDF) on each RDD in a D-stream. The *transformation* operators allow programs to produce a new D-stream from one or more parent streams. Unlike the MapReduce model, D-Streams support both *stateless* and *stateful* operations. The *Stateless* operations act independently on each micro-batch such as map and reduce. The *stateful* operations operate on multiple micro-batches such as aggregation over a sliding window.

D-streams provide stateful operators which are able to work on multiple micro-batches such as *Windowing* and *Incremental Aggregation*. The *Windowing* operator groups all records within a specific amount of time into a single RDD. For example, if a windowing is every 10 seconds and the sliding period is every one second, then an RDD would contain records from intervals [1,10], [2,11], [3,12], etc. As a result, same records appear in multiple intervals and the data processing is repeated. On the other hand, the *Incremental Aggregation* operator targets processing each micro-batch independently. Then, the results of multiple micro-batches are aggregated. Consequently, the *Incremental Aggregation* operator is more efficient than windowing operator because data processing is not repeated.

Spark supports fixed and sliding windowing while collecting data in micro-batches. It processes data based on its time of arrival. Spark uses the windowing period to collects as much data as possible to be processed within one micro-batch. Therefore, the back pressure can affect the job latency. Spark streaming enables to limit the number of collected records per second. As a result, the job latency will be enhanced. In addition, Spark streaming can control the consumption rate based on the current batch scheduling delays and processing times, so that the job receives as fast as it can process[2].

3.1.3 Parallel Recovery

D-streams use a recovery approach called *parallel recovery*. Parallel recovery periodically checkpoints the state of some RDDs. Then, it asynchronously replicates the checkpoints to different nodes. When a node fails, the recovery mechanism detects the lost partitions and launches tasks to recover them from the latest checkpoint. The design of RDDs simplifies the checkpointing mechanism because RDDs are read-only. This enables Spark to capture snapshots of RDDs asynchronously.

Spark uses the structure of RDDs to optimize capturing checkpoints. It stores the checkpoints in a stable storage. Checkpoints are beneficial for RDDs with wide dependencies because recovering from a failure can require a full re-execution of the job. In contrast, RDDs with narrow dependencies may not require checkpointing. When a node fails, the lost partitions can be re-computed in parallel on the other nodes.

3.2 Apache Flink

Apache Flink[3] is a high-level robust and reliable framework for Big Data analytics on heterogeneous data sets [25, 26]. Flink engine is able to execute various tasks such as machine learning, query processing, graph processing, batch processing, and stream processing. Flink consists of intermediate layers and an underlying optimizer that facilitates building complex systems for better performance. It is able to connect to external sources such as HDFS [27] and Apache Kafka. Therefore, it facilitates

[2] http://spark.apache.org/docs/latest/configuration.html
[3] https://flink.apache.org/

processing data from heterogeneous data sources. Furthermore, it has a resource manager which manages cluster's resources and collects statistics about the executed and running jobs. Flink consists of multiple layers, which will be discussed later on, to achieve its design goals. In addition, it relies on a lightweight fault tolerant algorithm which helps to enhance the overall system performance.

3.2.1 Flink Architecture

The Flink stack [25] consists of three layers: Sopremo, PACT, and Nephele, as depicted in Figure 3.1. These layers aim to translate the high-level programming language to a low-level one. Each layer represents a compilation step that tries to optimize the processing pipeline. Each layer comprises of a set of components that have certain responsibilities in the processing pipeline.

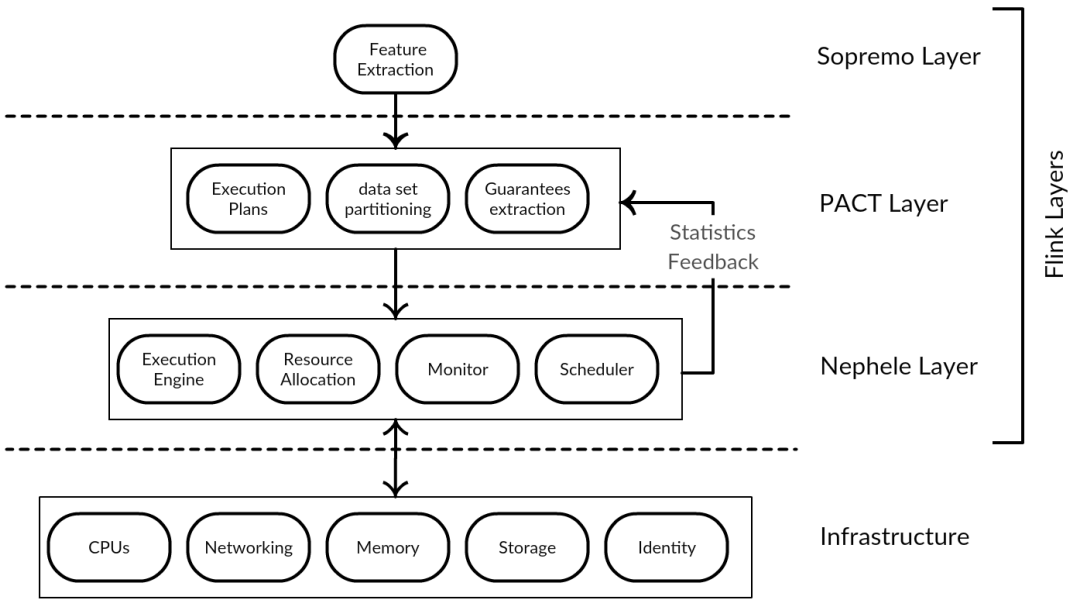

Figure 3.1: Flink stack

The Sopremo layer is responsible for operator implementations and information extraction from a job. It relies on an extensible query language and operator model called *Sopremo* [28]. This layer contains a compiler that enables extracting some properties from the submitted jobs at compile time. The extracted properties help in optimizing the submitted job in the following layers. A Sopremo program consists of logical operators in a DAG. Vertices represent tasks and edges represent the connections between the tasks. The Sopremo layer translates the submitted program to an operator plan. A PACT program [29] is the output of the Sopremo layer and the input to the PACT layer.

The PACT programming model is an extension to the MapReduce model called *PACTs* [30]. The PACT layer divides the input data sets into subsets according to the degree of parallelism. *PACTs* are responsible for defining a set of guarantees to determine which subsets will be processed together. Processing subsets are based on first-order functions which are executed at run time. Similar to MapReduce model, the user-defined first-order functions are independent of the parallelism degree. In addition to the MapReduce features, PACTs can form complex DAGs.

The PACT layer contains a special cost-based optimizer. According to the guarantees defined by the PACTs, this layer defines different execution plans. The cost-based optimizer is responsible for choosing the most suitable plan for a job. In addition to the statistics which are collected by Flink's resource manager, the optimizer decisions are influenced by the properties sent from the Sopremo layer. PACTs are the output of the PACT layer and the input to the Nephele layer.

The Nephele layer is the third layer in the Flink stack. It is the Flink's parallel execution engine and resource manager. This layer receives data flow programs as DAGs and a certain execution strategy from the PACT layer that suggests the degree of parallelism for each task. Nephele executes jobs on a cluster of worker nodes. It manages the cluster's infrastructure such as CPUs, networking, memory, and storage. This layer is responsible for resource allocation, job scheduling, execution monitoring, failure recovery, and collecting statistics about the execution time and the resource consumption. The gathered statistics are used in the PACT optimizer.

3.2.2 Streaming Engine

Flink has a streaming engine[4] to process and analyze real-time data, which is able to read unbounded partitioned streams of data from external sources called *DataStreams*. A *DataStream* aggregates data into time-based windows. It provides flexible windowing semantics where windows can be defined according to the number of records or a specific amount of time. *DataStreams* support high-level operators for processing data such as joins, grouping, filtering, and arithmetic operations.

When a stream processing job starts running on Flink, operators of *DataStreams* become part of the execution graph in DAG. Each task is composed of a set of input channels, an operator state, a user-defined function (UDF), and a set of output channels. Flink pulls data from input channels and executes the UDF to generate the output. If the rate of injected data is higher than the data processing rate, a back pressure problem will appear[5].

A data streaming pipeline consists of data sources, a stream processing job, and sinks to store results. In the normal case, the streaming job processes data at the same rate at which data is injected from the sources. If the stream processing job is not able to process data as fast as the injected data, the system could drop the additional data or buffer it somewhere. Data loss is not acceptable in many streaming

[4] https://ci.apache.org/projects/flink/flink-docs-master/apis/streaming/index.html

[5] http://data-artisans.com/how-flink-handles-backpressure/ - How Flink handles back-pressure

systems such as billing systems. Also, buffered data should be made in a stable storage because this data needs to be replayed in case of failure to prevent data loss.

Flink guarantees that each record is processed *exactly once* [26]. Therefore, it relies on buffering additional data in a durable storage such as Apache Kafka [21]. Flink uses distributed queues with bounded capacity between different tasks. An exhausted queue indicates that the receiver task is slower than the sender task. Therefore, the receiver will ask the sender to slow down its processing rate. If the sender is a source task, it will inject data in a slower rate to fulfill the requirements of the upcoming tasks.

Flink supports fixed and sliding windowing to process data streams. It can process data based on either the arrival time or the creation time of the injected data. Flink streaming engine can buffer records for a period of time before sending them downstream for processing. By default it buffers records for 100 milliseconds before processing them in bulk[6]. However, this value can be increased to enhance the overall performance of a job execution.

Flink uses the heap memory size to buffer records. It has two running modes that affect the allocated memory to each task[7]: *batch mode* and *streaming mode*. The *batch mode* starts each job with a pre-allocated amount of memory for each operator. The *streaming mode* starts running a job without allocating a predefined amount of memory. Instead, it tries to maximize the usage of the heap size dynamically.

3.2.3 Asynchronous Barrier Snapshotting

Flink streaming is a distributed stateful stream processing engine that aims to process large-scale continuous computations. It targets to process stream of data with high throughput and low latency. In stream processing frameworks, fault tolerance mechanism has an impact on message processing guarantees and the execution time. These frameworks can recover from failures by taking periodic snapshots of the execution graph. A snapshot is a global state of the execution graph that can be used to restart the computations from a specific state. It may consist of the records in transit, the processed records, and the operators' state.

There are many techniques to capture snapshots such as synchronous global snapshotting and asynchronous global snapshotting. Synchronous global snapshotting, which is used in [31], stops the system execution to collect the needed information to guarantee exactly-once semantics. Collecting such information impacts the system performance because it makes the system unavailable while capturing a snapshot. Asynchronous global snapshotting collects the needed information without stopping the system execution. Asynchronous Barrier Snapshotting (ABS) algorithm [26] follows the asynchronous global snapshotting technique.

The ABS algorithm aims to collect global periodic snapshots with minimal overhead on system resources in low latency and high throughput. It requires two

[6] https://ci.apache.org/projects/flink/flink-docs-release-0.8/streaming_guide.html

[7] https://ci.apache.org/projects/flink/flink-docs-release-0.8/cluster_setup.html

properties for each snapshot to guarantee correct results after recovery: termination and feasibility. Termination guarantees that the snapshot algorithm terminates in a finite time as long as all processes are alive. Feasibility guarantees that the snapshot itself contains all the needed information.

The ABS algorithm eliminates persisting the state of the channels while collecting a global snapshot due to the nature of executing jobs on Flink engine. Flink divides each job into tasks which are executed in order based on a DAG. The collected information of each task in the same execution order implies the state of input and output channels. Consequently, the snapshot size and the needed computations become less. The ABS algorithm injects special barrier markers in input data streams to ensure the execution order.

The barrier markers are pushed throughout the execution graph down to the sinks. The ABS algorithm uses the job manager in the Nephele layer as a central coordinator for pushing barriers periodically to all sources. When a source task receives a barrier, it takes a snapshot of its current state. Then, it broadcasts the barrier to all its output channels. When a non-source task, which depends on other tasks receives a barrier, it blocks the input channel that the barrier comes from. When a task receives barriers from all its input channels, it takes a snapshot of its current state. Then, the task broadcasts the barrier to its output channels and it unblocks all its input channels. After unblocking all input channels, the task continues its computation. The global snapshot consists of all snapshots from sources to sinks. The ABS algorithm considers some assumptions about the execution graph:

- All channels respect FIFO delivery order and can be blocked and unblocked.
- Tasks can trigger operations on their channels such as block, unblock, and send messages.
- All output channels support broadcast messages.
- Messages are injected only in the source tasks, which have zero input channels.

The ABS algorithm guarantees termination and feasibility. Termination is guaranteed based on the reliability of channels and DAG properties. The reliability of channels guarantees that tasks receive barriers eventually as long as they are still alive. The DAG properties guarantee that barriers are pushed from sources to sinks in order. On the other hand, the ABS algorithm guarantees feasibility because the global snapshot only represents the history of processed records. Furthermore, FIFO ordering delivery guarantees the order of input data and barriers. Therefore, the feasibility is guaranteed through the properties of channels.

The ABS algorithm is based on directed acyclic graphs. Therefore, it would not terminate when there are cycles in the execution graph. A task would be waiting forever for a barrier from one of its input channels that is an output channel of one of the upcoming tasks. In addition, the computation could be stopped because of blocking one of the input channels forever. Consequently, the termination and the feasibility will not be guaranteed.

The ABS algorithm extends the basic algorithm to allow cyclic graphs. It categorizes the edges, in the execution graph, into two categories: regular-edge and back-edge. The back-edge category implies that an input edge of a specific vertex is an output edge to an upcoming vertex. Back-edges are defined statically before graph execution. The ABS algorithm works as described before regarding the regular-edge category. When a task that has back-edges receives barriers from all its regular-edges, the task stores its local state. Then, the task starts logging all records that are coming from back-edges. Additionally, it broadcasts the barrier to its output channels. The task keeps receiving records from only its back-edges until receiving barriers from all of them. Then, the task combines its local state and the records which are in transit within the cycle. The combination forms the snapshot of this task. After taking the snapshot, the task unblocks all its input edges.

The modified ABS algorithm for cyclic graphs guarantees termination and feasibility. Termination is guaranteed because each task eventually receives barriers from all its input edges. Moreover, it avoids the deadlock because of marking the back-edges. Feasibility is guaranteed because each task state includes information about its local state and input channels. Each task considers the records in transit in presence of cyclic graphs. Furthermore, feasibility is guaranteed because of FIFO channel guarantees.

Flink uses the latest global snapshot to restart the whole execution graph when a failure happens. Each task uses its snapshot state as an initial state. Then, each task recovers its backup log, processes all its records, and starts ingesting from its input channels. For exactly-once semantics and with the guarantees of FIFO channels, the ABS algorithm [26] claims that duplicate records can be ignored when message sequence numbers are added from the sources. As a result, tasks can ignore messages with sequence numbers less than what they have already processed.

3.3 Lambda Architecture

The batch processing model has proven its accuracy in data processing. However, it lacks in producing results in a low latency manner. On the other hand, the stream processing model has proven the ability to produce results in low latency. Nonetheless, the subsequent model may produce inaccurate results, in some cases because it does not consider already processed data while processing new data. The Lambda architecture [2] combines the benefits of both processing models to deliver accurate results in low latency.

The Lambda architecture is a series of layers, each of which is responsible for some functionalities that are built on top of the layers underneath. It consists of five layers: batch, serving, speed, data, and the query layer. The batch layer stores data sets in an immutable state. It performs arbitrary functions of the stored data sets. The batch layer indexes its outcome in views, which are called batch views, to facilitate answering the expected questions, which are called queries. The serving layer loads the batch views and allows random reads. The speed layer ensures that the answers of queries include the result of processing recent data as quickly as needed.

The speed layer has another set of views. These views contain results of the data that is not included in the batch views yet. The data and query layers are the architecture's interfaces with external systems. The data layer is responsible for receiving new data. The query layer is responsible for receiving queries and responding with answers.

3.3.1 Batch Layer

The master data set is the most important part that has to be stored safely from corruption because it is the heart of the Lambda architecture. This architecture follows the *Write-Once-Read-Many (WORM)* paradigm to store the master data set. It distributes and replicates the master data set across multiple computers. The batch layer implements one of the re-computation algorithms to process data. A re-computation algorithm performs multiple processing iterations on the same data set. In each processing iteration, the batch layer processes the master data set as one batch.

The batch layer uses the batch processing model because the master data set has already been collected before data processing. It targets to produce accurate results. This layer can take a long time to process the master data set and generate the results accordingly. Therefore, the batch layer has a large latency in the order of minutes to hours for processing data. It stores the results in batch views.

The batch layer is a distributed system to store and process data. Thus, it is subject to the CAP theorem. It allows writes for new immutable data. The coordination among different write operations is not needed because each operation is independent. Similar to the batch processing model, each data item is processed exactly once. Thus, the batch layer chooses consistency over availability when network partitions.

3.3.2 Serving Layer

The serving layer uses the batch views to collect the outcome of a query. The process of answering a query combines a high latency task, which executes in the batch layer, and a low latency task, which executes in the serving layer. The serving layer is potentially serving outdated data due to the high latency of the batch layer. The Lambda architecture distributes the serving layer among several computers for data availability. Moreover, it creates indexes on the batch views to load them in a distributed manner. While designing indexes, latency and throughput are important performance factors.

The serving layer is responsible for read operations and answering already known queries. Therefore, this layer can be optimized to have the linked data on the same computer to minimize the latency and to increase the throughput. Each query type can have a separate view to store the needed data on the same computer. As a result, the system will have redundant data and it will consume more resources. In contrast, it will have higher throughput and lower latency because the architecture minimizes the communication among different computers. This is a known problem in relational databases about data normalization versus data denormalization [32]. The serving

layer follows the denormalization technique through data redundancy to avoid joins among different tables or documents.

The serving layer stores the batch views in a scalable database which is able to store data of arbitrary sizes. The database is a batch writable because the batch layer produces all views from scratch on each processing iteration. This database provides random reads via indexes which provide random access to small portions of the views. It does not provide random writes because the serving layer is designed to answer queries only. Thus, the database is simpler than other distributed databases that provide both random reads and writes. The simplicity of the database helps the serving layer to answer queries with low latency.

The serving layer is a distributed system and it is thus subject to the CAP theorem. The batch views do not contain the results of recent data. Therefore, this layer is neither consistent nor eventually consistent because it will always be out-dated. Thereby, the serving layer chooses availability over consistency when network partitions.

3.3.3 Speed Layer

The speed layer is responsible for low latency updates and satisfying the final requirements of the real-time systems. This layer is based on incremental computation algorithms, which process only the recent data. As running jobs over a complete data set is a resource intensive operation and time consuming, the speed layer aims to process the new data that the serving layer does not include. Therefore, the speed layer processes data which is significantly smaller than the master data set.

Once the serving layer includes the new produced data, the speed layer discards this data from its views. This layer should return results in real time. Thus, it does not consider the master data set while processing the recent data. As a result, the speed layer is more prone to errors due to inaccurate results or network congestion. However, the errors, in this layer, are short-lived because the batch layer will correct them in the next processing iteration.

Building such a layer requires two major functionalities. Firstly, the layer should only process the new data to produce real-time views. Secondly, it should store the real-time views in an optimal way to answer queries in low latency. There is a key difference between serving layer views and speed layer views. The serving layer views contain the results of processing almost the master data set. The speed layer views contain the results of recent data that the batch layer does not serve yet.

The speed layer can process the new data via one of two approaches: re-computation algorithms or incremental algorithms. Re-computation algorithms, as in the case of the batch layer, run a function over all the recent data, after which the old real-time views are replaced. Incremental algorithms run a function over the new data that the speed layer views do not include. Thereafter, the results are appended to the existing real-time views. Each approach has its strengths and weaknesses. Re-computation approach facilitates the creation of real-time views. Similar to the serving layer, the database does not need to consider random writes operations. Moreover, the results become more accurate because all recent data is

considered. On the other hand, incremental approach needs a database that supports random reads and writes to allow updating the views and answering the queries. Re-computation approach has a larger latency than incremental approach because the recent data might be big according to the needed amount of time for each processing iteration in the batch layer.

The batch layer can consist of multiple jobs if it needs a long time to process the master data set. A job can run once per month to compute the data from the first offset. While another job can run once per week to compute the remaining data which the first job does not include yet. As a result, the speed layer will have less data to process and it will have lower latency. For example, the speed layer will process a collected data during a week instead of a month. In addition, the output becomes more accurate because the speed layer can be based on approximation algorithms[8]. Approximation algorithms are used to find approximate results to minimize the required time for computing accurate results.

The computation of the speed layer and its views are distributed among different computers. Therefore, this layer is subject to the CAP theorem. It does not aim to have accurate results but approximate real-time results. Furthermore, it aims to support answering queries with the recent results in low latency. As a result, the speed layer chooses availability over consistency when network partitions.

3.3.4 Data and Query layers

The data layer is the system interface to receive new data. This layer should feed the batch layer and the speed layer with the new data. Thus, it should enable continuous and real-time processing of the incoming data. The data layer should provide the data at low latency to the speed layer. Also, it should send the data ordered and only once to both processing layers to prevent inaccurate results.

The data layer should be reliable enough to store the recent data that the serving layer does not include because the batch layer can take a long time to process the master data set. The speed layer calculates its views using either re-computation approach or incremental approach. Therefore, the data layer should support providing the speed layer with the recent data from the first offset when needed.

The query layer is the system interface to receive queries and answer them based on the processed data. Based on each question, it collects the required data from different views, then it returns the answer. This layer is query-dependent because the answer of some queries is the aggregation of the results from different views, whereas the answer of other queries needs processing of the results from different views.

The data and query layers can be distributed among different computers. If a system expects to receive a large number of requests, it is better to replicate these layers on different computers for higher throughput. Replicating the data layer helps in receiving and processing the new data faster. In addition, the recent data becomes distributed on many computers. Similarly, replicating the query layer helps in serving a larger number of client requests. Therefore, both layers are distributed and they

[8] https://en.wikipedia.org/wiki/Approximation_algorithm

choose availability over consistency when network partitions. As the data and query layers are distributed systems, they are thus subject to the CAP theorem. They choose availability over consistency when network partitions.

3.3.5 Lambda Architecture Layers

Figure 3.2 depicts how the Lambda architecture layers work together to answer different kinds of queries. The Lambda architecture has two layers for data processing: batch and speed layers. When a new record comes, the data layer receives and stores it in a temporary buffer. Then, it forwards this record to both processing layers. This record will be removed once the batch layer has processed it. The batch layer stores new records in the master data set. Once the speed layer receives a new record, it processes this record, after which it updates or replaces the real-time views accordingly. When there is a question to be answered, the query layer receives it and collects the needed information from the views of serving and speed layers. Then, it processes the results, if needed, and responds with the answer.

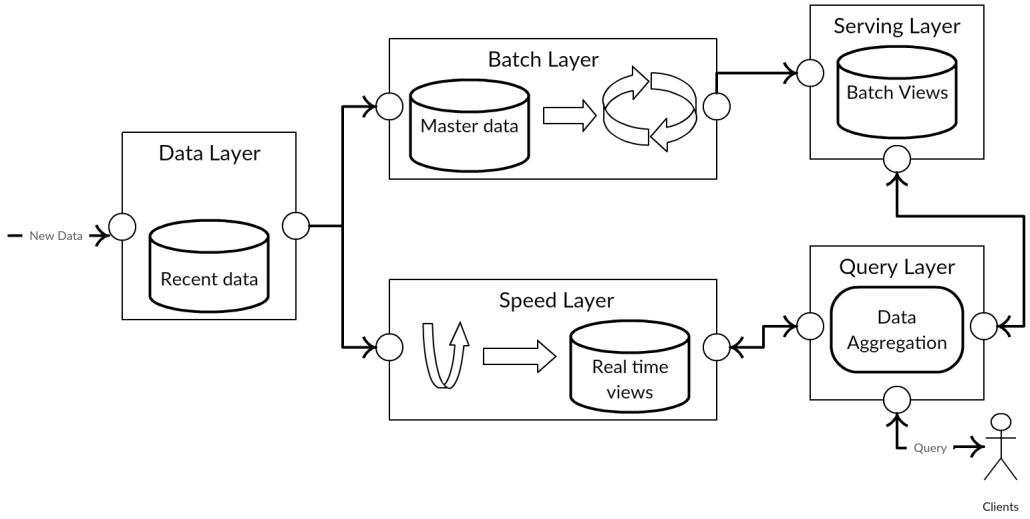

Figure 3.2: Lambda architecture

3.3.6 Recovery Mechanism

Lambda architecture layers can fail at any point of time. Each layer is distributed among many computers. The architecture should consider the failure of any computer. If one of the servers that is running the data layer fails, the other servers can tolerate this failure. Also, a new server can start running and receiving new data. However, portion of the recent data, which is distributed among servers of the data layer, may be lost. This should not be a problem because the lost data is already stored in the master data set. The batch layer will process this portion of data in the next processing iteration.

The speed and batch layers are processing layers. If the speed layer fails, it can recompute the recent data, which is stored in the data layer, and update its views accordingly. On the other hand, if the batch layer fails while processing the master data set, it can re-execute the failed job, similar to any batch processing job. However, the master data set must be stored and replicated to tolerate computer failures. Moreover, the batch views, in the serving layer, should be replicated because it can cause the system to be unavailable till generating new batch views.

3.4 Apache Beam

Apache Spark and Flink streaming engines are scalable and fault tolerance. They process Big Data with high throughput and low latency based on time-based windowing. However, Apache Spark and Flink streaming engines lack in providing windowing based on different factors such as sessions, categories, and users. In addition, they rely on watermarks to group records into data windows. A watermark is a system's notion of when all records in a certain data window is expected to be arrived. Apache Spark and Flink streaming engines emit a data window after a specific number of records or amount of time.

Due to the nature of distributed systems, watermarks are insufficient because they have two major outcomes with respect to correctness: fast or slow. When a watermark is fast, late records arrive behind the watermark. As a result, these records will be considered in upcoming data windows. A watermark can be slow because it can be held back for the entire pipeline by a single slow record. Therefore, watermarks have a limitation in processing data based on the event time instead of the processing time.

Apache Beam[9] is an open-source project based on the Google Cloud Dataflow model [33]. This model is built based on FlumeJava batch engine [34] and MillWheel streaming engine [35]. Apache Beam is a single unified framework that allows for a relatively simple expression of parallel computing in a way that is independent of the underlying execution engine. It provides the ability to control the amount of latency and correctness for any specific problem domain. Apache Beam allows for calculations of unbounded and un-ordered data sources based on the event time. Moreover, it separates the logical notion of data processing from the underlying physical implementation. Therefore, Apache Beam allows batch, micro-batch, or streaming processing using a unified programming model. The streaming engine groups unbounded data streams into data windowing. Apache Beam supports three types of windows to deal with unbounded data: fixed, sliding, and sessions. This framework processes each record separately in order to process it into the correct data window.

[9] https://github.com/apache/incubator-beam

3.4.1 Google Cloud Dataflow Architecture

The Cloud Dataflow model considers each record as a pair of *{key, value}*. The record's key is optional unless it is required by specific operations. The Cloud Dataflow model has two core transforms for data processing: *ParDo* and *GroupByKey*. The *ParDo* transform is used for generic parallel processing. This transform sends each input record to a UDF to be processed independently. For each input record, the UDF returns a zero or more output records. This mechanism facilitates processing unbounded data streams. The *GroupByKey* transform is used for key-grouping pairs. It collects all records for a given key before sending them for processing. The *GroupByKey* transform can deal with bounded data but it cannot process unbounded data without data windowing. Therefore, the Cloud Dataflow model uses *GroupByKeyAndWindow* operation to process unbounded data.

Windowing considers each record as 4-tuples of *{key, value, event-time, window number}*. Different windows can hold the same record but with a different window number. In fixed windows, records are assigned to a single window. While in sliding windows and sessions, records can be assigned to multiple windows. Merging data windows is needed in order to group them by a specific property such as the key. In addition, data windows need a mechanism to determine when the collected records should be emitted. The Cloud Dataflow model uses both *windowing* and *triggering* mechanisms to provide multiple answers, which are called panes, for any given window in low latency. The *windowing* determines where in the event time data is grouped together for processing. The *triggering* determines when in processing time the results are emitted as panes.

Triggering mechanism addresses the limitation of *watermarks* by enhancing the correctness of *GroupByKeyAndWindow* results. This mechanism provides three accumulation modes to control how multiple panes for the same window are related to each other: *discarding*, *accumulating*, and *accumulating and retracting*. The *discarding* mode targets to have independent panes for the same window. Upon triggering, the late records are processed and stored without any relation with old panes. The *accumulating* mode aims to replace old results with the new ones. The new pane includes the results of the old pane and the new results. This mode is similar to the Lambda architecture when new batch views replace the old ones. The *accumulating and retracting* mode follows the same semantics as *accumulating* mode. In addition, this mode does not replace the old panes. It keeps storing the old panes because they are needed in case a retraction operation is fired. Upon triggering, the contents of a data window are processed and the results are stored in a persistent state. If there is a pipeline with multiple serial *GroupByKeyAndWindow* operations, the panes of a single window may end up on separate keys when grouped downstream. Therefore, this kind of pipelines requires retractions.

3.4.2 Processing Guarantees

The Cloud Dataflow model provides *exactly once* processing guarantees [35]. It relies on a unique key of each record. Keys are the primary abstraction for aggregation

and comparison among different records. The model provides *a key extraction* UDF to assign a key to each record. Each record is stored in a highly available persistent store such as Bigtable [36] or Spanner [37]. When the model receives a new record from a client, it performs many steps to guarantee that each record is processed exactly once. Firstly, the model checks whether the record is duplicated, based on the record's key, to be discarded. Secondly, the record is processed based on a UDF. Thirdly, The results are stored in a persistent state. For better performance, the model collects multiple records to be stored instead of one-by-one. Finally, the model acknowledges the sender that the record has been processed.

A client may not receive the acknowledgement due to computer failures or network issues. As a result, the client retries to send the same record again. The Cloud Dataflow model identifies the duplicate records by their keys. It realizes that the acknowledgement of this record has been lost. Therefore, the model re-sends the acknowledgment without processing the record again. By this way, the Cloud Dataflow model guarantees *exactly once* processing semantics. However, if a key can be duplicated for multiple records, the model will not be able to differentiate between them.

The Cloud Dataflow model maintains a Bloom filter[10] of known records to provide fast and efficient checking against deduplication data, instead of checking all stored records for each new record. A Bloom filter is memory-efficient probabilistic data structure that is used to check whether a record is a member of a set. If a Bloom filter answers that a record does not exist, then the Cloud Dataflow model does not need to read the persistent store. On the other hand, if the Bloom filter answers that a record may exist, the model has to read the persistent store to determine whether the record is a duplicate. The model cleans the Bloom filters periodically to minimize the disk seeks.

3.4.3 Strong and Weak Production Mechanisms

The Cloud Dataflow model provides two approaches to capture checkpoint: *strong production* and *weak production*. The strong production approach checkpoints produced records before sending the acknowledgment in the same atomic write transaction. Therefore, if a failure happens before acknowledging a record, the model will roll-back all changes which are made in this transaction. The Cloud Dataflow model assumes that UDFs are idempotent. Therefore, executing same operations on same records, in case of failure, will not cause different states. If processing a record needs many sequential transformations, a slow transformation may slow down the overall performance. Figure 3.3 shows how the overall performance is affected by a slow transformation.

The weak production approach attempts to enhance the overall performance. It avoids having atomic transactions. Figure 3.4 shows how the overall performance is enhanced. If a transformation process discovers that the next one is a slow process, it checkpoints the state of the communication between them. Then, this

[10] http://en.wikipedia.org/wiki/Bloom_filter

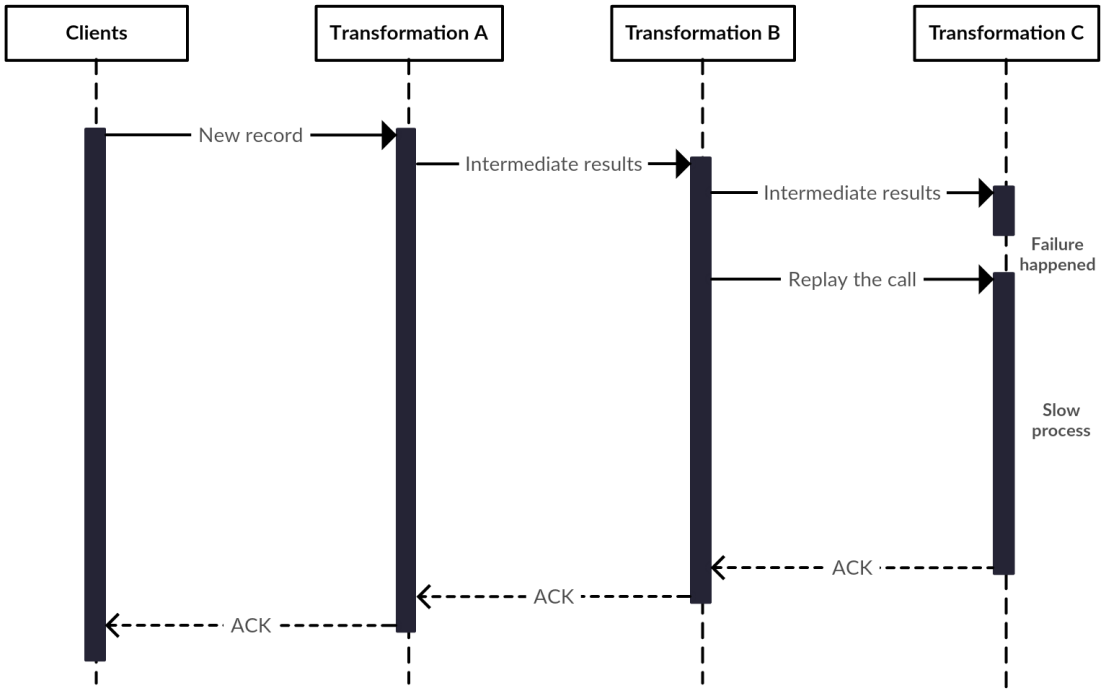

Figure 3.3: Dataflow strong production mechanism

transformation sends the acknowledgement to the sender transformation. As a result, the first transformation will not wait for a long time to get the acknowledgment. The checkpoints are deleted after receiving the acknowledgement.

3.4.4 Apache Beam Runners

Google has published the Cloud Dataflow API model as an open-source project[11] to enable other run-time engines to use the underlying architecture. The Google Cloud Dataflow[12] provides full integration with Google cloud services such as BigQuery [38], BigTable [36], and Compute Engine [39]. Google cloud services have to be running of Google infrastructure, therefore clients have to pay to use these services. Apache Beam uses the Cloud Dataflow model with different run-time engines called *runners*.

Apache Beam is a unified model to perform batch and stream processing pipelines on distributed processing engines such as Apache Spark and Flink[13]. This model enables performing same pipelines on different runners with minimal code changes. It supports four different runners: *DirectPipelineRunner*, *DataflowPipelineRunner*, *FlinkPipelineRunner*, and *SparkPipelineRunner*. *DirectPipelineRunner* enables users to run the pipeline on a local machine. *DataflowPipelineRunner* executes the pipeline on Google Cloud Dataflow. *FlinkPipelineRunner* enables users to execute pipelines

[11] https://github.com/GoogleCloudPlatform/DataflowJavaSDK
[12] https://cloud.google.com/dataflow/
[13] https://github.com/apache/incubator-beam

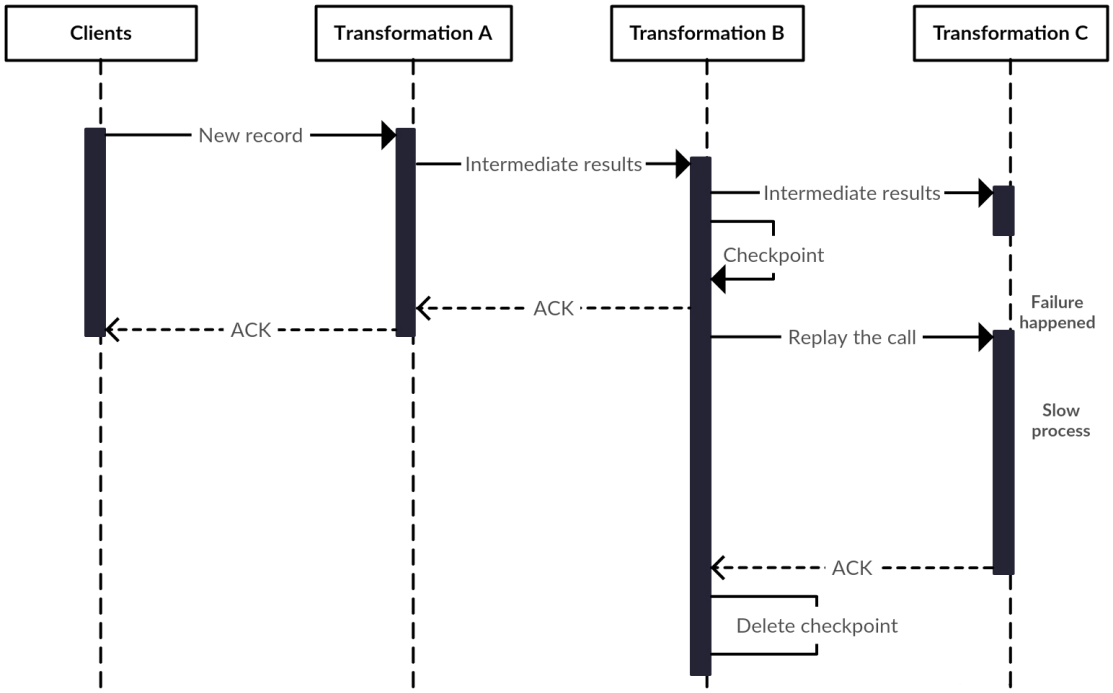

Figure 3.4: Dataflow weak production mechanism

on a Flink cluster. *SparkPipelineRunner* enables users to execute pipelines on a Spark cluster. Apache Beam is still on the implementation phase. Therefore, functionalities of the Cloud Dataflow model are not fully supported by Spark and Flink runners.

Chapter 4

Comparative Analysis

Apache Spark, Apache Flink, and Apache Beam are different methodologies to process large-scale unbounded data sets. They attempt to process data with high throughput and produce results in low latency. Each framework has a different approach to deal with data streams. In addition, each framework has its strengths and weaknesses. Apache Spark is an extension to the MapReduce model. It collects records from unbounded data streams for a period of time, after which it processes them as bounded data sets in small batches. In contrast, Apache Flink processes data at its time of arrival. Once a record is received, Flink sends this record downstream for processing and producing the results accordingly.

Apache Spark and Flink do not consider already processed data while processing the recent data. Therefore, they might produce inaccurate results if the already processed data influences the results of recent data. Additionally, these frameworks could process the same record multiple times because they focus on producing real-time results more than producing accurate results. The Lambda architecture addresses the limitations of the stream processing frameworks. This architecture combines the batch processing model, to produce accurate results, and the stream processing model, to produce real-time results. Apache Beam is a unified framework for processing batch and stream data. It is able to deal with different data windowing. Furthermore, it can consider the old results after producing the new results.

4.1 Windowing Mechanism

Apache Spark collects records into data windows before sending them downstream for processing. Figure 4.1 shows how Spark stream processing engine deals with data streams. Firstly, it collects records for a period of time, according to the windowing period through a separate stage. Secondly, it processes the collected data in bulk through one or more stages, based on the needed processing. Finally, it stores the produced results through a final stage. Apache Spark can process records based on the arrival time of each record.

Apache Flink processes records at their time of arrival. Figure 4.2 depicts how Flink stream processing engine processes each record. A Flink stream processing

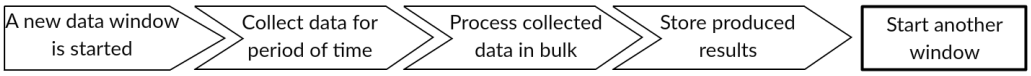

Figure 4.1: Spark stream processing pipeline

pipeline consists of multiple tasks. It starts with a source task, which injects new records, and ends with a sink task, which stores the results in external systems. There are one or more processing tasks between source and sink tasks. Each task has a queue to buffer the received records. The processing tasks process the records and produce the results accordingly. The produced results are buffered and grouped with old results of the same window, if needed. After the end of a data window, the buffered results are stored. Apache Flink supports data windowing based on the arrival time and the creation time of each record. However, it relies on watermarks to emit data windows. Once a watermark is fired, the late records will not be considered in the correct data window.

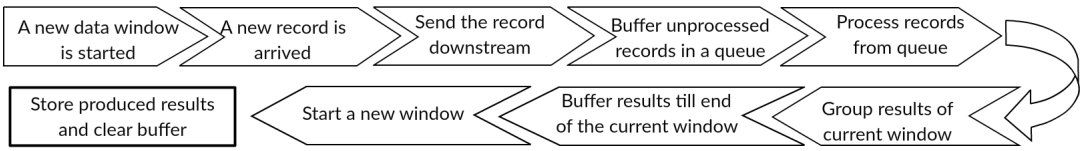

Figure 4.2: Flink stream processing pipeline

The Lambda architecture consists of two different processing pipelines: batch and stream processing. The batch processing pipeline does not have a windowing mechanism because it processes the master data set in bulk. This pipeline can produce results in groups based on specific parameters such as time of arrival or time of creation. Although, the results will be produced in a high latency fashion. On the other hand, the stream processing pipeline is implemented using a stream processing engine such as Apache Spark and Apache Flink. Therefore, this pipeline depends on what the implemented stream processing engine supports.

Apache Beam is able to process data based on the creation time and the arrival time of each record. Figure 4.3 depicts the windowing mechanism in Apache Beam. When a new record is received, Apache Beam checks it against deduplication. If the record is duplicated, the model sends an acknowledgment without any processing. Otherwise, the record is processed and its results are stored in the correct pane. According to the creation time of each record, the model determines whether this record belongs to the current data window or an old one. After firing a trigger of a data window, its pane is stored in a persistent storage. The accumulation mode determines the action that will be performed after firing a trigger of an old data window.

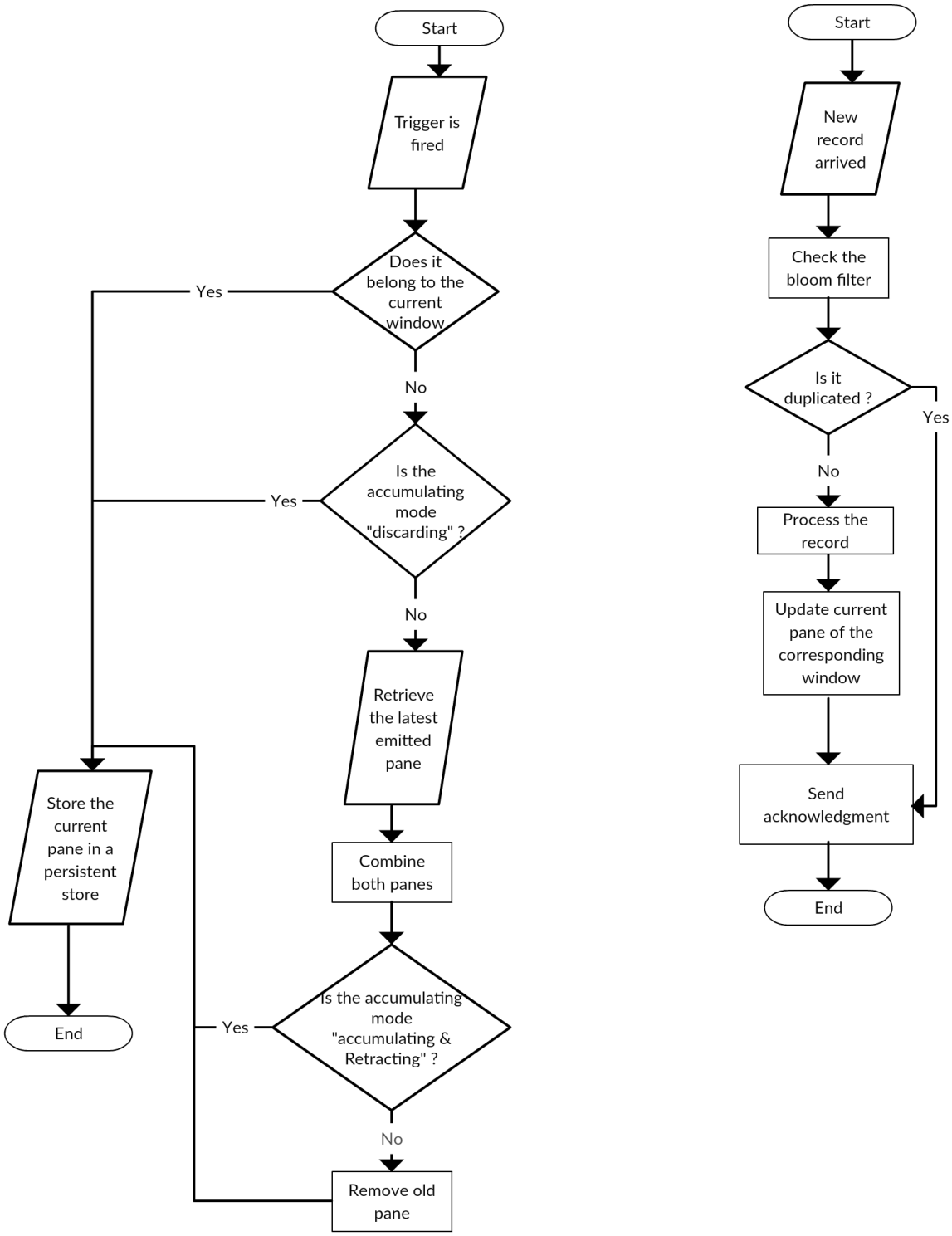

Figure 4.3: Apache Beam stream processing pipeline

Apache Spark, Apache Flink, and Apache Beam support different types of data windows. Table 4.1 describes which types are supported by each framework. All of them support fixed and sliding windows. Apache Spark and Apache Flink are not able to support session windows. The Lambda architecture can support session windows based on the batch layer. However, the results of each session will be produced in high latency. Apache Beam supports session windows. This framework triggers the end of a session when no more records are received for a period of time.

Feature	**Apache Spark**	**Apache Flink**	**Apache Beam**
Fixed Windows	✓	✓	✓
Sliding Windows	✓	✓	✓
Session Windows	✗	✗	✓

Table 4.1: Available windowing mechanisms in the selected stream processing engines

4.2 Processing and Result Guarantees

Stream processing frameworks target to provide guarantees for data processing and results accuracy. Table 4.2 shows the guarantees that are provided by each framework. Apache Spark processes each record *exactly once*. Furthermore, it stores the acknowledged records in a persistent storage. If a job fails during the final stage, which is responsible for storing the results, Spark will start the failed stage from the beginning. However, it will not process the acknowledged records. It can guarantee that results are stored *exactly once* when either idempotent updates or Transactional updates is supported.

Apache Flink guarantees processing records *exactly once* based on the ABS algorithm. If a job fails, Flink will use the latest global snapshot to restore the local state of each task. However, the records that are processed after the latest global checkpoint will be processed again. Therefore, Flink guarantees *at least once* semantic for the produced results. The Lambda architecture can guarantee *exactly once* semantics for both data processing and stored results because the batch layer processes the master data set in bulk. However, the accurate results are produced in a high latency fashion.

Apache Beam guarantees processing records *exactly once* based on the data window that each record belongs to. Additionally, it can guarantee *exactly once* semantic for the produced results as long as each record has a unique id. This model manages a Bloom filter to track the processed records. If a failure happens or same record is received again, Apache Beam checks the Bloom filter to make sure that this record is not processed before.

[1]Exactly once semantics are guaranteed when either idempotent updates or Transactional updates is used - http://spark.apache.org/docs/latest/streaming-programming-guide.html#semantics-of-output-operations

[2]Exactly once guarantees could be achieved with idempotent updates

Guarantee	Apache Spark	Apache Flink	Apache Beam
Exactly Once data processing	✓	✓	✓
Exactly Once for producing results	✓[1]	✗[2]	✓[3]

Table 4.2: The accuracy guarantees in the selected stream processing engines

4.3 Fault Tolerance Mechanism

Apache Spark divides each stream processing job into micro-batches. Each micro-batch comprises multiple stages. Spark periodically checkpoints the state of RDDs with wide dependencies. When a stage fails, Spark engine recovers the latest checkpoint to prevent re-executing the whole micro-batch. It launches tasks to recover the required RDDs from the latest checkpoint. Spark will recomputes the RDDs with narrow dependencies using the *parallel recovery* mechanism.

Apache Flink uses the ABS algorithm as a recovery mechanism. The ABS algorithm collects global periodic snapshots with minimal overhead on system resources in low latency and with high throughput. When a job fails, Flink uses the latest global snapshot to restart the job. Each task, in the failed job, uses its snapshot state as an initial state. Then, each task recovers its backup log, processes all its records, and starts ingesting from its input channels.

The Lambda architecture has two different processing systems which can fail while processing records. In the batch layer, if a job fails, it can be restarted similar to any batch processing job. If a job fails in the speed layer, there are two options to follow. Firstly, the speed layer could follow the recovery mechanism of the implemented stream processing framework. Secondly, the speed layer can re-process all the recent data, which is stored in the data layer, and produce new views accordingly.

Apache Beam has two approaches to recover failure: *strong production* and *weak production*. The *strong production* approach considers an atomic transaction for each record. If processing a record fails, the model re-processes the same record. This framework assumes that UDFs are idempotent. Therefore, performing same operations on same records, in case of failure, will not cause different states. However, this model might have performance issues because it depends on sequential transformations. The *weak production* approach captures a checkpoint in case of slow transformations. In case of failures, it recalls the same transformation based on the captured checkpoint.

4.4 Strengths and Weaknesses

Apache Spark stream processing engine is based on the batch processing model to provide its guarantees. It collects data into micro-batches for processing. In case of

[3]This mechanism is implemented in Google Cloud Dataflow runner

failure, Apache Spark re-runs the micro-batch, similar to the batch processing model. However, Spark may have a problem in case of back pressure if there is no limits for processing records per second and the back pressure handling is not enabled. As a result, it may deliver data in high latency. Furthermore, Apache Spark processes data according to its time of arrival. If Spark engine receives late records, it will consider these records in the current data window. Therefore, neither event-time data processing nor session windowing are supported.

Apache Flink collects records in data windows. It processes records independently at their time of arrival. Apache Flink produces the final results of each data window after processing a specific number of records or a specific amount of time. Therefore, it is able to produce results in real time. It relies on buffering data in a durable storage in case of back pressure. It accepts specific number of records to be processed within the duration of each data window. It is able to process data according to their time of creation or time of arrival. It uses watermarks to determine when to emit a data window. Flink stream processing engine fires a watermark either once or continuously based on a given time interval. If a watermark is fired once, some records might arrive behind the watermark. Thus, those late records will not be processed in the correct data window. On the other hand, if a watermark is fired continuously, this could be wasting of resources if there is no late records. Additionally, Flink is not able to process session windowing.

The Lambda architecture combines the benefits of the batch processing model and the stream processing model. It processes the master data set to deliver accurate results. Furthermore, this architecture is able to deliver results in real time. However, the Lambda architecture requires to manage two independent Big Data pipelines. The first pipeline uses the batch processing model to produce accurate results but in high latency. This pipeline considers the already processed data while processing recent data. The second pipeline uses the stream processing model to process the recent data and produce approximate results but in real time. The combination of the two pipelines enhances the accuracy of results and produces real-time results. However, such an architecture needs to build and maintain two different systems. Furthermore, it needs an accurate approach to merge the produced results from both systems.

Apache Beam is a unified model for both batch and stream processing. It supports time-based and data-based windowing. Moreover, it has different accumulation modes to correlate different results of the same data window with each other. Furthermore, it supports processing records based on their time of creation. In addition, this framework is able to process session windowing. It has different runners for processing data. Apache Beam can process data through Apache Spark, Apache Flink, or Google Cloud Dataflow pipeline.

Chapter 5

Experiments and Results

This chapter explains the experiments which are done on selected stream processing frameworks. Section 5.1 illustrates the experimental setup of the used cluster. Section 5.2 shows different Kafka configurations that are used. Section 5.3 illustrates different processing pipelines that are tested. While Sections 5.4, 5.5, and 5.6 explains the experimental results on Apache Spark, Apache Flink, and Apache Beam respectively. Section 5.7 discusses some comparisons of the tested frameworks on the basis of presented experimental results.

5.1 Experimental Setup

There are six virtual machines (VMs) which are used for the experiments. Each VM has four virtual central processing units (VCPUs) and 15,360MB of RAM. The VMs use *CentOS Linux 7* as an operating system. Three VMs run Apache Spark and Apache Flink. The other three VMs hold the raw data and the results. They run Apache Zookeeper, Apache Kafka, and HDFS. Table 5.1 shows the version of each framework which are used in these experiments.

Apache Spark	1.6
Apache Flink	1.0.0
Apache Beam	0.1.0-incubating-SNAPSHOT
Apache Kafka	0.9
Apache ZooKeeper	3.4.6
Hadoop	2.6.3

Table 5.1: The versions of the used frameworks

The following experiments are based on a *word count* pipeline for 10GB data set of tweets gathered from Twitter streaming API[1]. Data windowing of 10 seconds is used. A snapshot of the execution graph is taken each minute, if the fault tolerance is enabled. Each test case is executed five times and the median[2] result is documented.

[1] https://dev.twitter.com/streaming/overview
[2] https://en.wikipedia.org/wiki/Median

5.2 Apache Kafka Configuration

There are two Kafka configurations which are used for the experiments. Table 5.2 shows different configurations for producers and consumers. The first set is the default one. The second set is a fault tolerant configuration that makes Kafka as much reliable and consistent as possible[3].

Property	Default Configuration	Fault Tolerant Configuration
Producer acks	1	all
Block buffer when Full	false	true
Number of producer retries	0	MAX INT
Kafka Replication factor	0	At least 3
Minimal insync replicas	0	2
Auto commit consumer offsets	true	false
Unclean Kafka leader election	true	false

Table 5.2: Different Kafka configurations

The data set is stored in three different Kafka topics. Each topic is divided into three partitions. Different topics will be used for different test cases. Table 5.3 shows the different topics and how data is distributed among different partitions. Furthermore, it illustrates the key of records in each topic. The value of each record is a tweet. $Topic_A$ distributes the data set among its partitions in a skewed way. $Topic_B$ and $Topic_C$ divide the data set equally among their partitions. $Topic_A$ and $Topic_B$ use User Identifier(UID) as the key for each record. $Topic_C$ uses the creation time of each tweet as its key.

Topic Name	Data in $Partition_0$	Data in $Partition_1$	Data in $Partition_2$	Record's Key
$Topic_A$	21%	33%	46%	UID
$Topic_B$	33.33%	33.33%	33.33%	UID
$Topic_C$	33.33%	33.33%	33.33%	Time-stamp

Table 5.3: Different Kafka topics

5.3 Processing Pipelines

Different processing pipelines can influence the performance of the stream processing engine. This chapter examines three different stream processing pipelines: arrival-time, session, and event-time. Figure 5.1 presents the common stages among these pipelines. The processing pipelines start with reading records from Apache Kafka

[3] http://www.slideshare.net/gwenshap/kafka-reliability-when-it-absolutely-positively-has-to-be-there

and assigning these records to a data window. After processing each data window, they store the results back into Kafka. In this figure, connectors *A, B, C* refer to the arrival-time, session, and event-time processing pipelines, respectively.

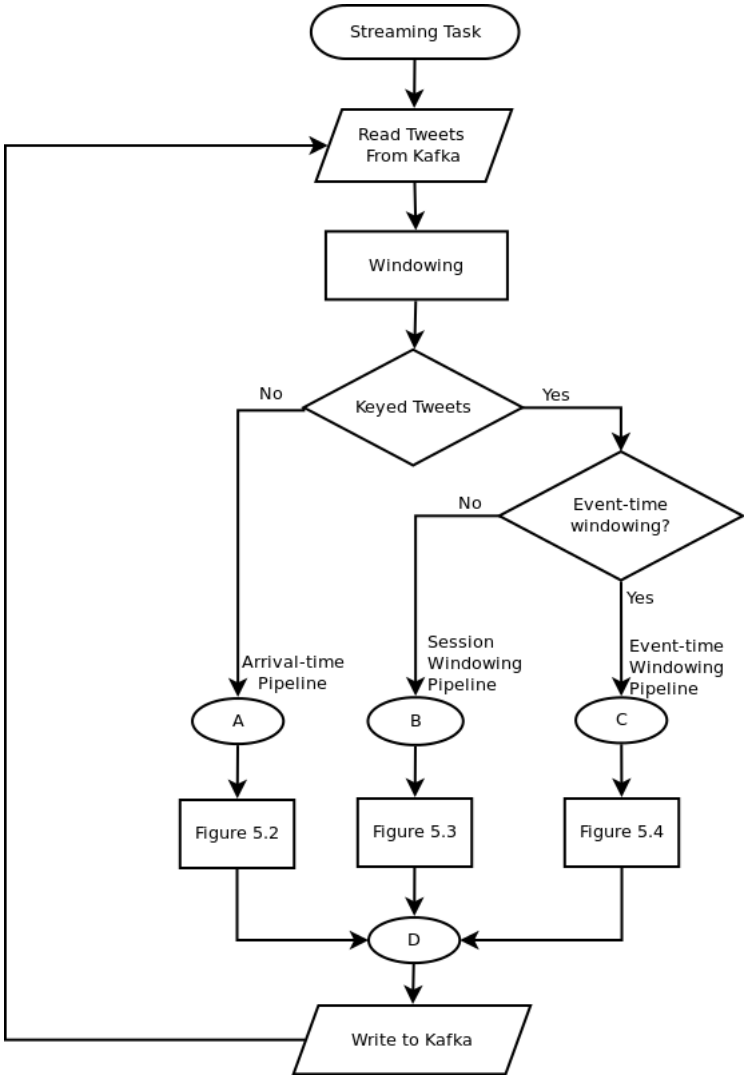

Figure 5.1: Processing pipeline

Figure 5.2 shows the straightforward approach to produce the number of word occurrences in each data window based on the arrival time of the records. When a processing pipeline receives a tweet, it splits the tweet into a set of words. It assigns a number for each word that represents the number of its occurrences in the tweet. Then, the processing engine collects each word and its number of occurrences from different tweets within a single computer to find out its number of occurrences in all tweets.

Figure 5.3 presents a pipeline to process tweets of a set of users. The session stream processing pipeline targets producing the number of word occurrences for each user independently. It has two different approaches. Firstly, it can group each

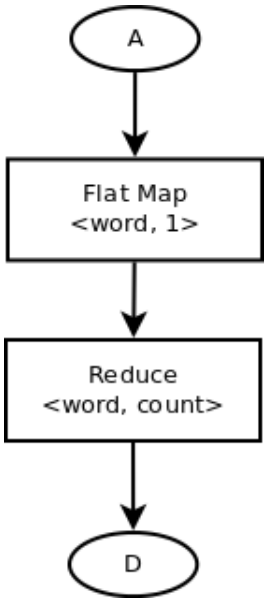

Figure 5.2: Arrival-time stream processing pipeline

user's tweets before processing, so that there will be one record for each user within each data window. This record contains the UID and a list of its tweets within a specific period of time. Secondly, it can split each tweet into a key-value pair of *{{UID,word}, 1}*, so that the key of each record will be a pair of UID and a word.

Figure 5.4 presents a pipeline to process tweets based on their time of creation. Similar to the session windowing stream processing pipeline, the event-time stream processing pipeline has two different approaches to process records. Firstly, it can group tweets based of their time of creation before processing them, so that there will be one record for each time period within each data window. This record contains a time and a list of its tweets which have been created within this time. Secondly, it can split each tweet into a key-value pair of *{{time,word}, 1}*, so that the key of each record will be a pair of a creation time and a word.

5.4 Apache Spark

Spark targets to maximize the number of records to process for each batch. The test cases in this section show how the batch size affects the overall execution time. They use time-based windows of 10 seconds. Therefore, Spark streaming engine collects records for 10 seconds, after which it processes them in one batch. The test cases expect real-time results because of the small windowing period. If the rate of incoming data is faster than the processing rate, Spark would end up delivering results in high latency. The following test cases do not enable back pressure support feature[4]. Sections 5.4.1, 5.4.2 and 5.4.3 use the arrival-time pipeline, which has

[4] https://databricks.com/blog/2015/09/09/announcing-apache-spark-1-5.html

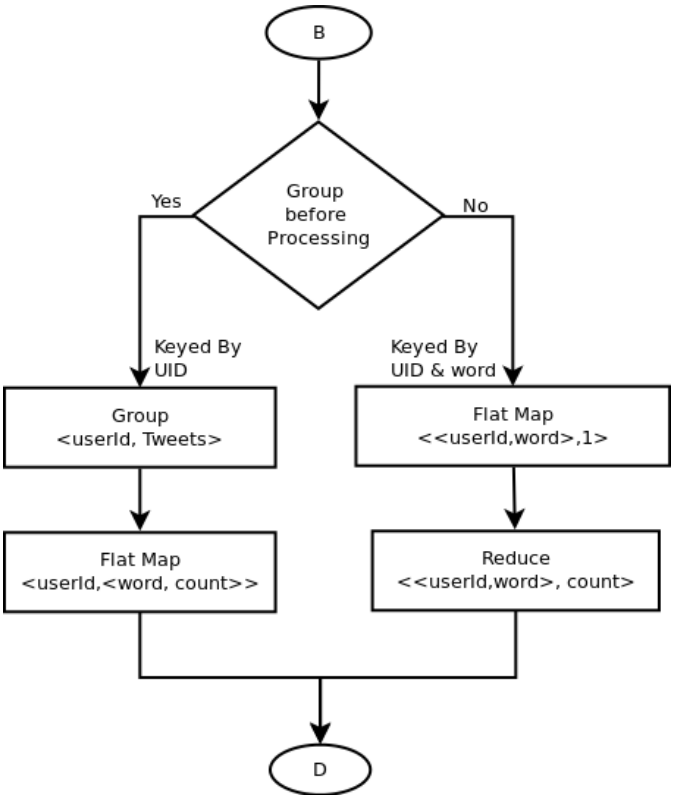

Figure 5.3: Session stream processing pipeline

already been explained in the Figure 5.2. Sections 5.4.1 and 5.4.2 use Kafka $Topic_B$ as an input stream, while Section 5.4.3 uses $Topic_A$ and $Topic_B$, which has been already discussed in Section 5.2. Section 5.4.4 discusses the execution time of the different pipelines.

5.4.1 Processing Time

The experiments illustrate how Spark deals with different data set sizes. These experiments limit the number of records to read per second from each Kafka partition. Figure 5.5 depicts the maximum number of records that each worker can read per second. This figure shows that there are eight experiments with different subsets of tweets to process per batch: 10GB, 6.7GB, 5GB, 3.3GB, 1.7GB, 0.91GB, 0.59GB, 0.3GB respectively. Apache Spark enables limiting the maximum receiving rate of records per partition through the *spark.streaming.kafka.maxRatePerPartition* property.

Spark has two types of receivers: *reliable* and *unreliable*[5]. The *reliable* receiver is used when data sources can send acknowledgements after storing data. The *unreliable* receiver is used when data sources cannot acknowledge storing data. In the following

[5] http://people.apache.org/~pwendell/spark-nightly/spark-master-docs/latest/streaming-programming-guide.html#input-dstreams-and-receivers

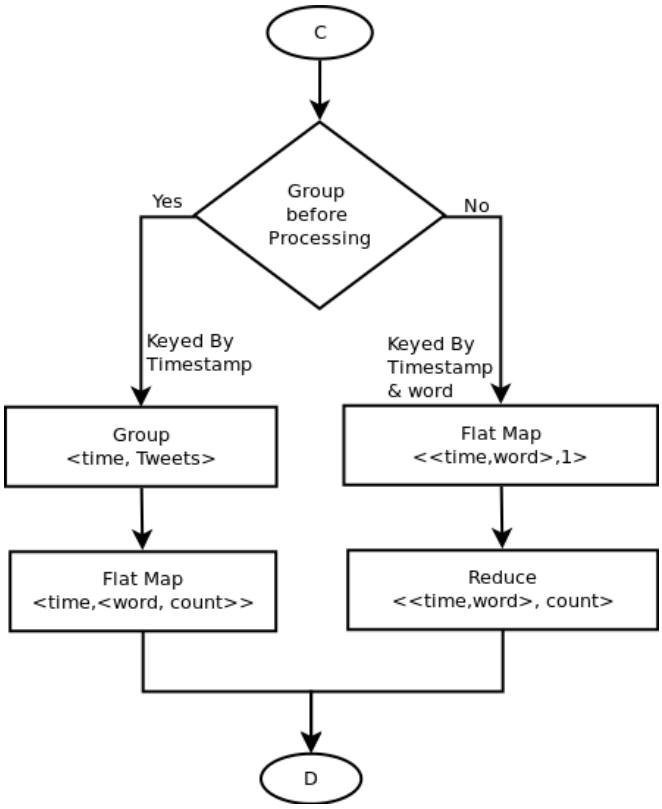

Figure 5.4: Event-time stream processing pipeline

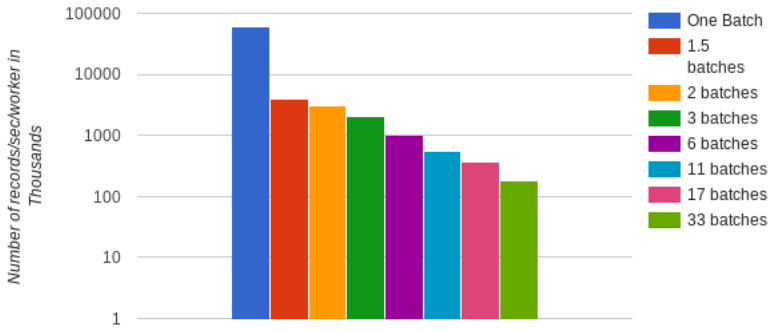

Figure 5.5: Number of processed records per second for different number of batches

experiments, the *reliable* receiver is used because Apache Kafka can acknowledge the stored records.

Figure 5.6 depicts the required execution time to produce the final results. This figure illustrates that Spark streaming engine collects as many records as possible to get the benefits of the batch processing model, unless there is a limit for reading records per second. Spark requires more time to produce results in real time. Figure 5.7

depicts the required amount of time for each batch in the eight experiments. For example, Spark will produce the first set of results after 3.6 minutes when processing 5GB per batch. While it will produce the first set of results after 36 seconds when processing 0.59GB per batch. Therefore, limiting the number of records to read per second controls the latency of producing results. Nonetheless, minimizing the number of records affects the overall execution time.

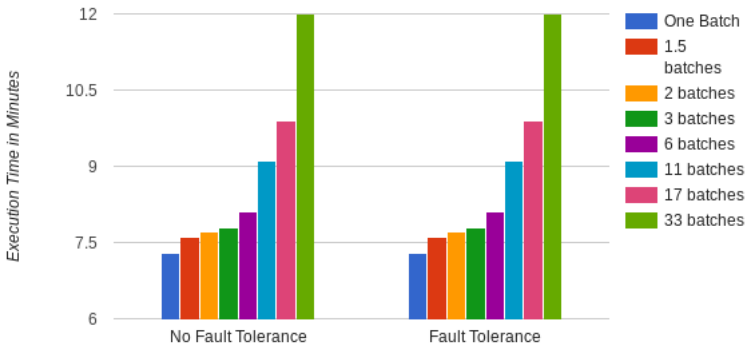

Figure 5.6: Spark execution time for different number of batches

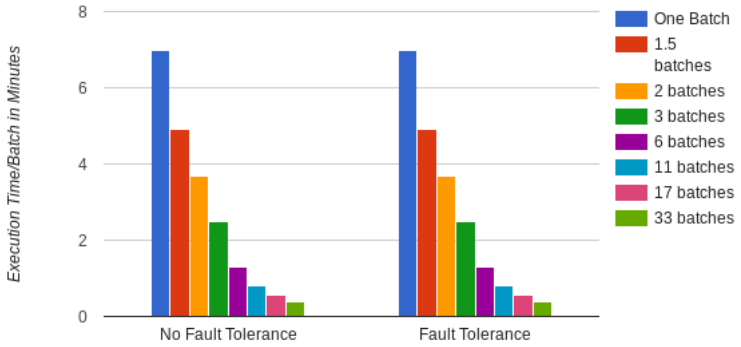

Figure 5.7: Spark execution time for each micro-batch

Both Figures 5.6 and 5.7 show the execution time when a snapshot is captured while processing the data. The fault tolerance mechanism shows that it does not affect the overall performance with the normal Spark configuration. However, Spark can prevent data loss on driver recovery. This can be enabled by setting the *spark.streaming.receiver.writeAheadLog.enable* property to *true*. This property enables to checkpoint the acknowledged records when the fault tolerance mechanism

is enabled. When a failure happens, the acknowledged records will not be processed again. Figure 5.8 presents the execution time after enabling this property. It shows that Spark takes more time to store the acknowledged records in HDFS.

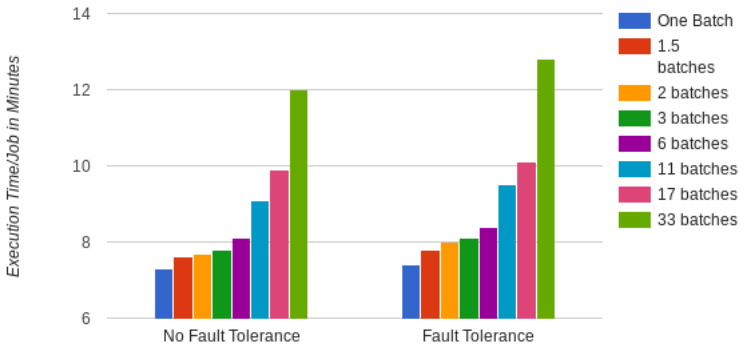

Figure 5.8: Spark execution time with *Reliable* Receiver

5.4.2 The Accuracy of Results

Spark divides each job into stages according to the required computations. It guarantees that each record will be processed *exactly once* because it processes subsets in batches[6], similar to the batch processing model. However, the accuracy of the produced results, which are stored in Kafka, is not well defined. Table 5.4 explains the accuracy of results based on the observations from the test cases in different failure scenarios when the *spark.streaming.receiver.writeAheadLog.enable* property is enabled.

Failure	**Spark + Default Kafka Configuration**	**Spark + Fault Tolerant Kafka Configuration**
The network between different brokers is partitioned	At Most Once	Exactly Once
The network between a broker and a worker is partitioned	At Least Once	At Least Once
Broker Failures	At Most Once	At Least Once
Worker Failures	At Least Once	At Least Once

Table 5.4: The accuracy of results with Spark and Kafka when failures happen

[6] http://spark.apache.org/docs/latest/streaming-kafka-integration.html

The accuracy of results is affected by different Kafka configurations, network partitions, and broker failures. In this section, network partitioning is implemented using a network proxy that prevents a specific server to send packets to another server. A broker failure is performed by shutting a server down. The network partitioning can take place between either different brokers or a broker and a worker. Figure 5.9 depicts how the network partitioning can happen in two different classes. A Kafka partition leader may not be able to communicate with other brokers due to network partitioning, however it can communicate with Spark workers. On the other hand, Kafka brokers can communicate with each other but one worker is unable to communicate with a specific broker.

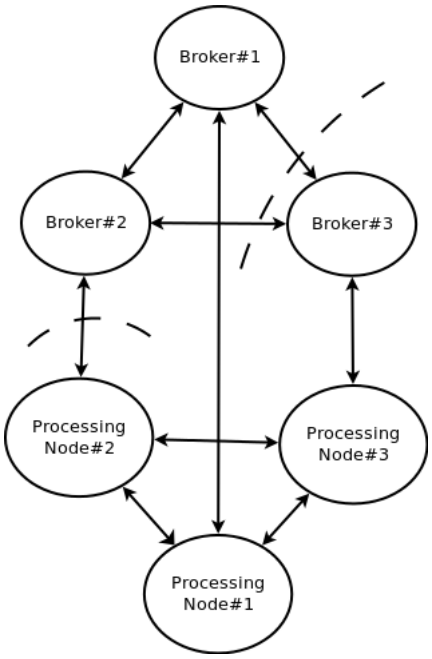

Figure 5.9: Network partitioning classes

Due to the network partitions, Broker#3 cannot communicate with other brokers for a period of time. However, Worker#3 is able to send messages to this broker. When the default Kafka configuration is used, Spark workers will receive acknowledgements once records are stored in one broker[7]. After a period of time, ZooKeeper will detect that Broker#3 is dead. Therefore, it will assign the leadership of that partition to another broker. Consequently, the acknowledged records from Broker#3 will be lost and the result accuracy will be *at most once*. On the other hand, when the fault tolerant Kafka configuration is used, workers will not receive acknowledgements before storing the records in all Kafka brokers. This leads to *exactly once* guarantees. The network can be partitioned and prevent, for example, Broker#2 and Worker#2 to communicate. This could happen while Worker#2 is waiting for an acknowledgement. In both Kafka configurations, Spark will re-send the same records to be stored.

[7] http://kafka.apache.org/documentation.html#producerconfigs

Therefore, this will lead to *at least once* guarantees.

A broker could fail before sending an acknowledgement of stored records. In this case, Spark will try to store the same records again. As a result, duplicate records will be exist. This leads to *at least once* guarantees. Similar to Kafka servers, a worker in Spark cluster could fail. This worker might fail after sending records to be stored and before receiving an acknowledgement. Consequently, another worker will try to store them again. This leads to *at least once* guarantees.

5.4.3 Different Kafka Topics

The data distribution on Kafka partitions affects the overall execution time of a job. Figure 5.10 depicts that if the data is distributed in a skewed way among different partitions of a topic, which has been already discussed in Table 5.3, the processing latency will be higher. Spark distributes the number of topic partitions on the existing workers. Therefore, one of the worker will take longer time because it is responsible for processing a partition that holds more data.

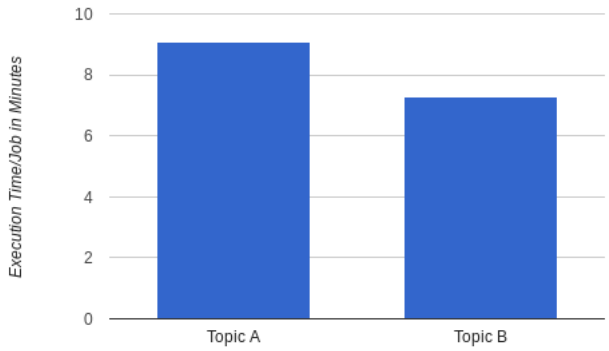

Figure 5.10: Spark execution time with different Kafka topics

5.4.4 Different Processing Pipelines

Spark does not start a new stage before the completion of previous stages to complete shuffling records. This behaviour affects the overall execution time of a stream processing job. Figure 5.11 depicts the execution time of different processing pipelines, which have already been discussed in Section 5.3. $Pipeline_A$ counts the number of occurrences for each word. $Pipeline_B$ counts the number of occurrences for each word written by each user. Based on the test cases, Spark throws *OutOfMemory* exception when trying to group all records of each user in a single worker. This problem happens when one of the tasks is too large[8]. While it takes 70 minutes

[8] http://spark.apache.org/docs/latest/tuning.html#memory-usage-of-reduce-tasks

to count the word occurrences for each user when processing tweets as a key-value pair of *{{UID,word}, 1}*. $Pipeline_A$ does not need to shuffle data through hard drives. While running this pipeline, each worker shuffles only 15.1 MB of records to be stored. In contrast, $Pipeline_B$ shuffles 5.5 GB of records through RAM and 600 MB of records through hard drives. In addition, each worker shuffles 770 MB of records to be stored.

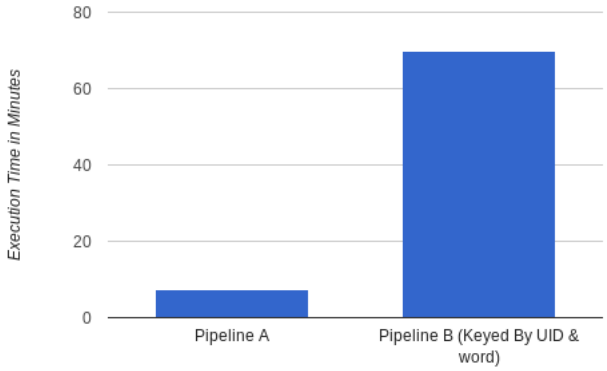

Figure 5.11: Spark execution time with different processing pipelines

5.5 Apache Flink

This section presents the test cases on Apache Flink according to different factors. Section 5.5.1 shows the impact of different running modes on the execution time of a job. Section 5.5.2 illustrates the impact of buffering data before processing. Section 5.5.3 discusses the impact of the ABS algorithm on the accuracy of results. Sections 5.5.4 and 5.5.5 illuminate the execution time with different Kafka topics and different processing pipelines, respectively.

5.5.1 Running Modes

Flink offers two running modes for a cluster: *batch* and *streaming*. Figure 5.12 depicts the execution time of both running modes with both Kafka configurations, which have already been described in Section 5.2. This figure is based on $Pipeline_A$, which has already been presented in Section 5.3. The figure shows the execution time when enabling fault tolerance and when disabling it. Both cases run without any failures.

The results show that the job latency is lower when the default Kafka configuration is used. Flink stores records in Apache Kafka asynchronously and it expects to receive an acknowledgement. Flink logs an error when the acknowledgement is not received within a specific amount of time[9]. Apache Kafka sends the acknowledgement when

[9] https://github.com/apache/flink/blob/release-1.0/flink-streaming-connectors/flink-connector-kafka-base/src/main/java/org/apache/flink/streaming/connectors/kafka/FlinkKafkaProducerBase.java

the records are stored in at least one broker, in case the default Kafka configuration is used. On the other hand, Apache Kafka sends the acknowledgement after storing the records in all replicas when the fault tolerant Kafka configuration is used. The fault tolerant Kafka configuration adds more overhead but it guarantees that all brokers are synchronized. Figure 5.12 shows that the fault tolerance mechanism has a small effect on the overall execution time. Furthermore, it depicts that the job latency of both running modes are the same. However, the impact of different running modes appears with different Buffer timeout values.

Figure 5.12: Flink execution time of both *batch* and *streaming* modes

5.5.2 Buffer Timeout

Flink is able to buffer pulled records at the source task before sending these records downstream for processing. The following experiments show how the performance is affected by different values of the buffer timeout in both Flink running modes. Figure 5.13 depicts the execution time with different buffer timeouts in the *batch* mode. This figure shows that using the full buffer will not cause any changes in the execution time because each job is limited by a specific amount of heap size. Additionally, it illustrates that if the source task sends each record once it is arrived, the job latency will be much higher. Figure 5.14 depicts the execution time with different buffers in the *streaming* mode. This figure shows that when the full buffer timeout is used, the job latency is enhanced. In the *streaming* mode, the heap size is allocated dynamically. Therefore, stream processing jobs can benefit from the available heap size.

5.5.3 The Accuracy of Results

Flink provides *exactly once* processing guarantees through its fault tolerance mechanism[10]. However, the accuracy of the produced results, which are stored in Kafka,

[10] http://data-artisans.com/kafka-flink-a-practical-how-to/

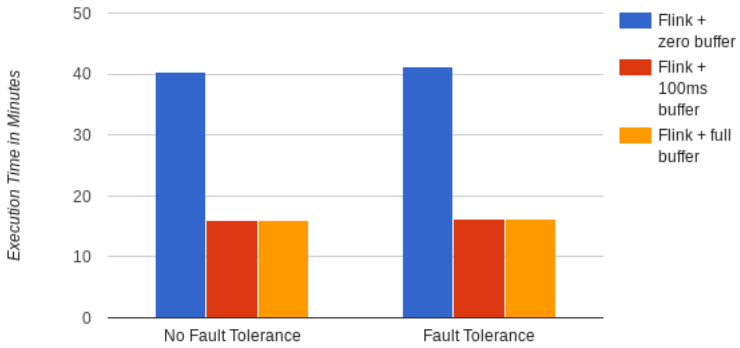

Figure 5.13: Flink execution time with different buffers in *batch* mode

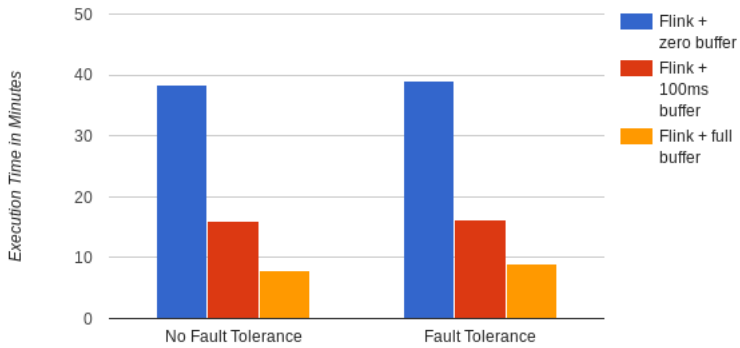

Figure 5.14: Flink execution time with different buffers in *streaming* mode

needs more clarifications. Table 5.5 explains the accuracy of results based on the observations from the test cases when different failure classes happen, as depicted in Figure 5.9. The experiments use the default and fault tolerant Kafka configurations. Through each experiment, a snapshot of the execution is taken every one minute. As described before in Section 5.5.2, the accuracy of results is affected by different Kafka configurations, the network partitioning, and broker failures.

Based on what has already been discussed in Section 5.4.2, Broker#3 cannot communicate with other brokers for a period of time. However, TaskManager#3 is able to send messages to this broker. When the default Kafka configuration is used, task managers will receive the callback once records are stored in one broker. After a period of time, ZooKeeper will detect that Broker#3 is dead. Therefore, it will assign the leadership of this partition to another broker. Consequently, the acknowledged records from Broker#3 will be lost and the accuracy of results will

Failure	Flink + Default Kafka Configuration	Flink + Fault Tolerant Kafka Configuration
The network between different brokers is partitioned	At Most Once	At Least Once
The network between a broker and a job manager is partitioned	At Least Once	At Least Once
Broker failures	At Most Once	At Least Once
Task Manager failures	At Least Once	At Least Once

Table 5.5: The accuracy of results with Flink and Kafka when failures happen

be *at most once*. On the other hand, when the fault tolerant Kafka configuration is used, task managers will not receive acknowledgements before storing the records in all replicas. If Flink does not receive the callback, an exception will be thrown and the current task will fail. Flink will use the latest global snapshot to recover the failed task. The stored records after the latest snapshot will be processed and stored again. This leads to *at least once* guarantees.

The network partitioning could happen between a task manager and a broker. For example, TaskManager#2 cannot communicate with Broker#2. As a result, the running task will fail and the latest global snapshot will be recovered. The same scenario happens when a task manager fails. In both Kafka configurations, the stored records after the latest snapshot will be processed and stored again. This leads to *at least once* guarantees.

Similar to task manager failures, a broker could fail before sending an acknowledgement of stored records. In this case, the running task will fail and the latest global snapshot will be recovered. When the default Kafka configuration is used, the acknowledged records will be lost. This will lead to *at most once* semantics. When the fault tolerant Kafka configuration is used, the acknowledged records are replicated but some of them will be processed again because they are stored after the latest snapshot. This leads to *at least once* guarantees.

5.5.4 Processing Time of Different Kafka Topics

Similar to Apache Spark as discussed in Section 5.4.3, the data distribution on Kafka partitions affects the overall execution time of a job. Figure 5.15 depicts that when the data is distributed in a skewed way among different partitions of a topic, which has been already discussed in Table 5.3, the processing latency will be higher. Flink distributes the number of topic partitions on the existing task managers. Therefore, one of those task managers will take longer time to process a partition that holds more data.

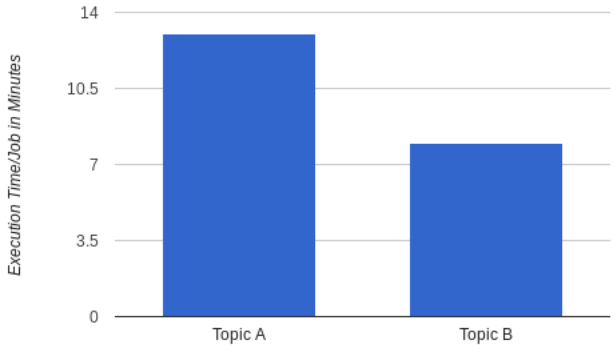

Figure 5.15: Flink execution time with different Kafka topics

5.5.5 Processing Time of Different Pipelines

Flink is able to start a next task before the completion of its predecessor task. It shuffles records while a task is running. This behaviour enhances the overall execution time of a stream processing job. Figure 5.16 depicts the execution time of different processing pipelines, which have already been presented in Section 5.3. Based on the test cases, Flink takes eight minutes to process $Pipeline_A$. Unlike Spark, Flink is able to collect all tweets of a single user on the same task manager. It takes 270 minutes to execute this job. When the records are processed as a key-value pair of *{{UID,word}, 1}*, it takes 25 minutes to count the word occurrences of each user.

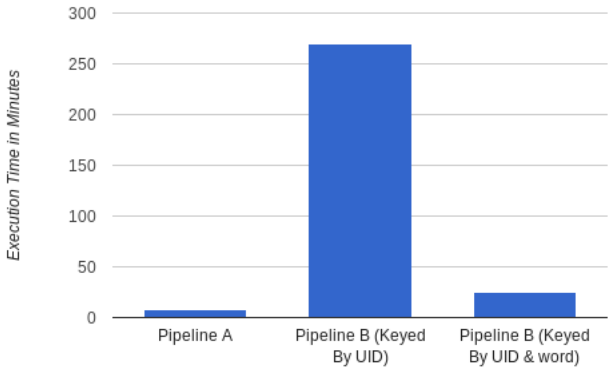

Figure 5.16: Flink execution time with different processing pipelines

Flink can process data based on its time of creation. It relies on watermarks to emit the results of a data window. There are two triggering mechanisms to fire watermarks: *EventTimeTrigger* and *ContinuousEventTimeTrigger*. The *EventTimeTrigger* mechanism fires a watermark only once, while the *ContinuousEventTimeTrigger*

mechanism continuously fires watermarks based on a given time interval[11]. Both triggering mechanisms are tested. Flink cluster crashes when the *ContinuousEventTimeTrigger* mechanism is used after processing almost half of the data set. Figure 5.17 depicts the execution time of $Pipeline_C$ when the *EventTimeTrigger* mechanism is used. The test cases prove that Flink is able to process data based on its time of creation. However, the last time for receiving late records should be known. Otherwise, Flink will create another data window for the late records with a new watermark.

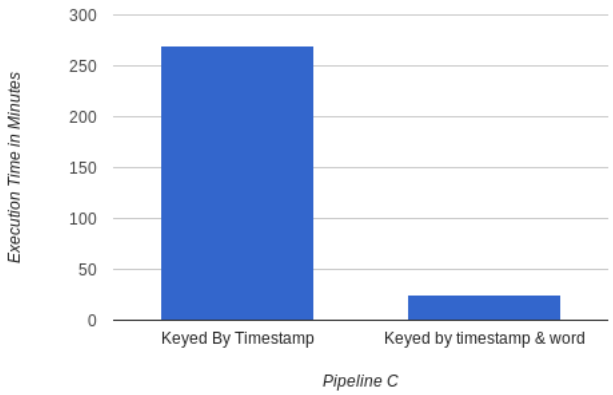

Figure 5.17: Flink execution time with event-time pipeline

5.6 Apache Beam

Apache Beam has three runners to perform stream processing jobs: Spark runner, Flink runner, and Google Cloud Dataflow runner. To execute a stream processing job using Google Cloud Dataflow runner, Google's infrastructure must be used. Therefore, it would be unfair to compare this runner with others. Section 5.6.1 explains the experiments that are done on Apache Spark cluster. Section 5.6.2 explains the experiments that are done on Apache Flink cluster based on the available features in Flink runner. The processing pipelines, which have already mentioned in Section 5.3, are used.

5.6.1 Spark Runner

$Pipeline_A$ is tested on Spark runner, however the current implementation can run Spark jobs on either a *YARN* cluster, similar to Apache Hadoop, or a single worker. After testing $Pipeline_A$ on a single Spark worker, it takes three hours to produce the results. Moreover, it shuffles 51GB of records on memory and 8.1GB on hard drives.

[11] https://ci.apache.org/projects/flink/flink-docs-master/api/java/org/apache/flink/streaming/api/windowing/triggers/package-summary.html

5.6.2 Flink Runner

Flink runner is able to process data based on its time of arrival, its time of creation, or sessions. $Pipeline_A$, $Pipeline_B$, and $Pipeline_C$ are tested. All pipelines can be executed on the Flink cluster. However, the cluster fails after processing half of the data set. Flink runner relies on *Continuous Triggers* which create a watermark for each data window. These watermarks are stored in memory and they are triggered continuously.

5.7 Discussion

Apache Spark and Flink have a different behaviour while processing data streams. Spark terminates micro-batches if there is no data to process. While Flink waits until the end of windowing period because it can process each record independently. Spark tries to maximize the number of records to be processed in each micro-batch, when the back pressure property is disabled. On the other hand, Flink processes a particular number of records to produce results by end of the windowing period. The number of processed records for each windowing period affects the required time to produce results.

Figure 5.18 depicts the required time to process an average number of records per second within each processing node. Spark shows a better performance than Flink while processing a large number of records per batch. However, Flink shows a better performance than Spark while producing results in low latency. Additionally, Flink processes the same amount of records when the windowing period is in the order of minutes. It buffers the produced results before storing them until the buffer is full or the current window ends. When no records exist to be pulled, Flink waits until the end of the windowing period. On average, Flink processes a smaller amount of records when the windowing period is in the order of minutes.

The data shuffling between processing nodes have an impact on the overall execution time. Apache Spark shuffles the results of a processing stage after the completion of this stage. Therefore, the following stage cannot start while the predecessor stage is still running. On the other hand, Flink shuffles results of a processing task once it starts to produce. Moreover, the next task can start its processing while the predecessor task is still running. Figure 5.19 depicts the required execution time for different processing pipelines when Spark and Flink are used. Spark shows a better performance when there are a small amount of records to be shuffled. On the other hand, Flink has a better performance while more data processing is required.

5.8 Future Work

This thesis explains the available functionalities of Apache Spark, Flink, Beam. It shows the limitations of each framework based on the used versions. Apache Spark is

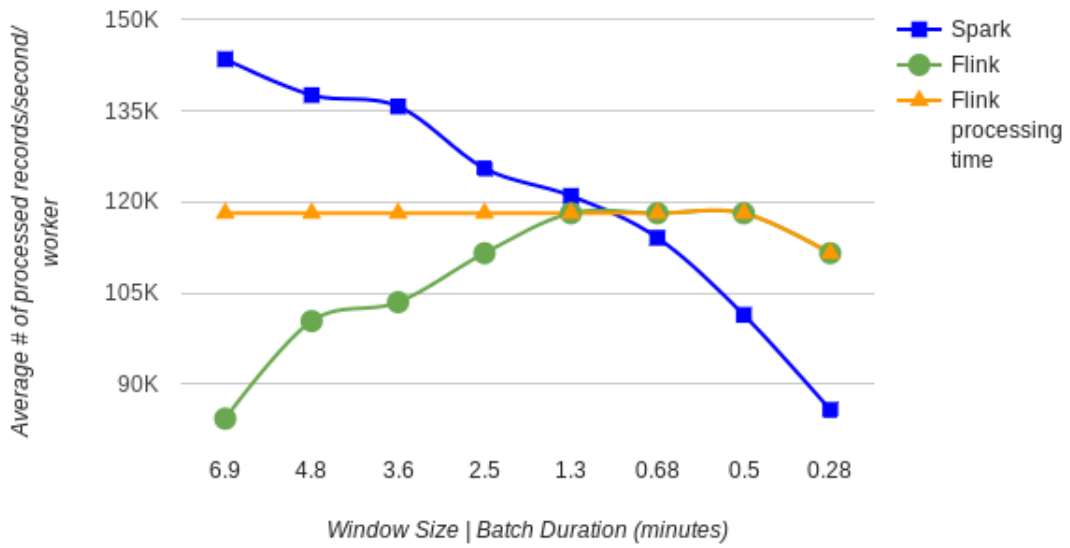

Figure 5.18: A comparison between Spark and Flink in different window sizes

going to release version 2.0[12] that manages cluster's memory more efficiently. It claims that the new version would be faster than version 1.6. Moreover, the new version targets to support session windowing. Furthermore, it uses SQL and Streamlined APIs to enhance the development speed. On the other hand, Apache Flink is going to release version 1.1.0[13] that supports session windowing. Both Spark and Flink try to be a stable runner in Apache Beam. Apache Beam is still in the development phase and it does not support all features of the Google Cloud Dataflow model. The first stable release of Apache Beam will be released during September, 2016[14].

An extension to what have already done in this thesis, is to test the new functionalities of each framework and find their strengths and weaknesses. The session windowing mechanism in both Spark and Flink should be tested and compared with the corresponding mechanism in the Google Cloud Dataflow model. Additionally, each runner, in Apache Beam, should be tested and different runners should be compared to know the similarities and differences.

[12] https://databricks.com/blog/2016/05/11/apache-spark-2-0-technical-preview-easier-faster-and-smarter.html

[13] http://data-artisans.com/session-windowing-in-flink/

[14] https://drive.google.com/open?id=17i7SHViboWtLEZw27iabdMisPl987WWxvapJaXg_dEE

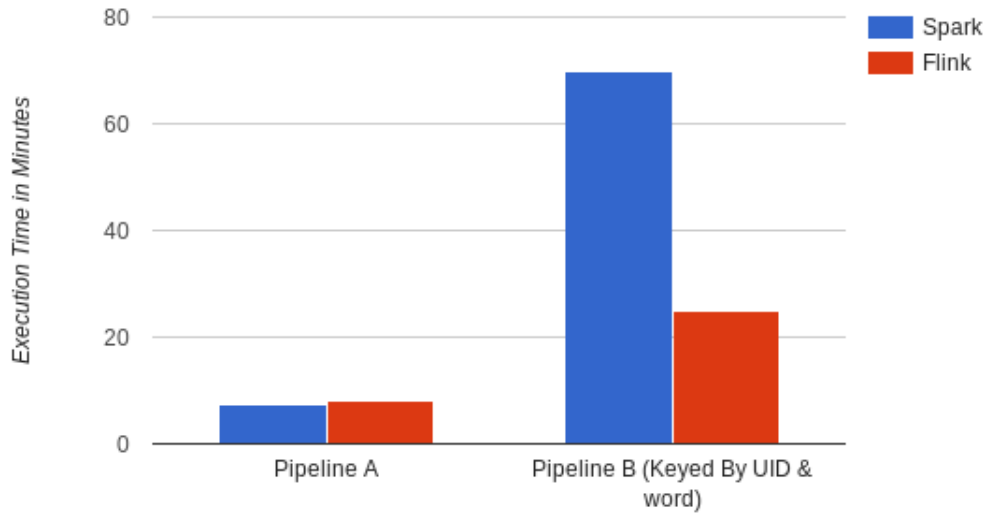

Figure 5.19: A comparison between Flink and Spark in different processing pipelines

Chapter 6

Conclusion

The main contribution of this thesis is to make a comparison between Apache Spark, Apache Flink, and Apache Beam according to the windowing and processing mechanisms, the fault tolerance techniques, the processing guarantees, and the accuracy of produced results. The conducted experiments have focused on examining the available functionalities of each framework.

Apache Spark divides a stream of data into small batches to process them. The experiments showed that Spark should be well-configured to deal with back pressure and to produce results with low latency. When the results are expected from each micro-batch within a given deadline, Spark streaming engine produced the results after the deadline because of the back pressure problem. After limiting the number of records to inject per second, Spark produced results with lower latency within the given time constraints. The experiments proved that Spark takes lower execution time to process the same data set when more records are processed within each micro-batch.

Spark captures snapshots from RDDs with wide dependencies, while it uses the *parallel recovery* mechanism to recover RDDs with narrow dependencies from failures with minimal network overhead. The experiments showed that the fault tolerance mechanism has a minor effect on the execution time. Spark requires more time to store the acknowledged results in a persistent storage. Although, this mechanism enhances the accuracy of produced results. The experiments presented that Spark can guarantee *exactly once* semantics for the produced results in a specific scenario when the fault tolerance Kafka configuration is used with the absent of worker failures. When the default Kafka configuration is used, the experiments showed that Spark guarantees *at most once* semantics.

Spark processes data at its time of arrival. It lacks in processing data at its time of creation or based on user sessions. The experiments demonstrated that different processing pipelines have different execution time to produce identical results. Spark failed to process jobs when one of the tasks has too large data to process. However, the experiments found that the same results can be obtained when the processing pipeline is designed in a better way to minimize the size of data chunks to be processed for each task.

Apache Flink has two running modes: *batch* and *streaming*. The experiments showed how running modes with different buffering periods affect the overall processing time for the same data set. Flink offered the maximum possible performance when the *streaming* mode and the full buffer timeout are used. The experiments proved that the full buffer timeout does not have impact when the *batch* mode is used because each task has a fixed amount of memory to use.

Apache Flink uses the ABS algorithm to collect global periodic snapshots asynchronously with minimal overhead on system resources with low latency and high throughput. The experiments demonstrated that the ABS algorithm has a small impact on the overall performance and it affects the accuracy of results. The experiments indicated that Flink guarantees *at least once* semantics for the produced results when the fault tolerance Kafka configuration is used. Furthermore, it guarantees *at most once* when the default Kafka configuration is used.

Apache Flink can process records at their time of arrival or time of creation. It lacks in processing records based on session windowing. The experiments illustrated that Flink can process data at its time of creation when the *EventTimeTrigger* is used. In contrast, Flink cluster crashed when the *ContinuousEventTimeTrigger* was used because Flink manages a watermark for each data window in the memory. When late records arrive after a data window is emitted, Flink creates another data window for the late records with a new watermark.

Apache Beam is able to process data at its time of arrival or time of creation. This framework is still on the early development stages. Spark and Flink runners are tested during the experiments. It was observed from the experiments that the current implementation of SparkRunner uses only one worker node to process a data set. The experiments illustrated that FlinkRunner relies on *ContinuousEventTrigger* to accumulate multiple panes of the same data window. Similar to the experiments performed on Flink, the Flink cluster in this case crashed after processing a few GBs of the data set.

The conducted experiments on Apache Spark and Apache Flink demonstrated the following observations:

- Spark showed a better performance than Flink while processing a large number of records per batch, whereas Flink showed a better performance than Spark while producing results in low latency.

- Spark should be well-configured to deal with back pressure while Flink usually produced the results by the end of each windowing period because it injects a specific amount of records to be processed for each data window.

- Flink has almost the same execution time as Spark when the *streaming* mode and the full buffer timeout are used.

- Spark was not able to perform the processing pipelines that require too large data to be processed for each task. On the other hand, Flink can perform different pipelines but with higher latency.

- Spark can provide *exactly once* guarantees for the produced results when the fault tolerance Kafka configuration is used with the absent of worker failures. Further, Flink can only provide *at least once* guarantees for the produced results when the fault tolerance Kafka configuration is used.
- The fault tolerance mechanism in Apache Spark and Apache Flink has a small impact on the execution time.
- Data distribution on different partitions of a Kafka topic influences the execution time of Apache Spark and Apache Flink.
- Flink cannot process data sets with many late records when the *Continuous Triggers* are used.

Bibliography

[1] T. White, *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (4. ed., revised & updated)*. O'Reilly, 2015.

[2] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, no. 2007, p. 21, 2007.

[5] V. Kalavri and V. Vlassov, "Mapreduce: Limitations, optimizations and open issues," in *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2013 / 11th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA-13 / 12th IEEE International Conference on Ubiquitous Computing and Communications, IUCC-2013, Melbourne, Australia, July 16-18, 2013*, pp. 1031–1038, 2013.

[6] X. Liu, N. Iftikhar, and X. Xie, "Survey of real-time processing systems for big data," in *18th International Database Engineering & Applications Symposium, IDEAS 2014, Porto, Portugal, July 7-9, 2014*, pp. 356–361, 2014.

[7] B. Ellis, *Real-time analytics: Techniques to analyze and visualize streaming data*. John Wiley & Sons, 2014.

[8] M. Stonebraker, U. Çetintemel, and S. B. Zdonik, "The 8 requirements of real-time stream processing," *SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.

[9] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10, IEEE, 2010.

[10] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[11] E. A. Brewer, "Pushing the CAP: strategies for consistency and availability," *IEEE Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[12] G. Gottlob and M. Y. Vardi, eds., *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings*, vol. 893 of *Lecture Notes in Computer Science*, Springer, 1995.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 205–220, ACM, 2007.

[14] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[15] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[16] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pp. 305–319, 2014.

[17] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*, pp. 245–256, 2011.

[18] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[19] L. Lamport, "Paxos made simple, fast, and byzantine," in *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002*, pp. 7–9, 2002.

[20] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, 2010.

[21] J. Kreps, N. Narkhede, J. Rao, *et al.*, "Kafka: A distributed messaging system for log processing," NetDB, 2011.

[22] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "Semantics and evaluation techniques for window aggregates in data streams," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pp. 311–322, 2005.

[23] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pp. 15–28, 2012.

[24] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: fault-tolerant streaming computation at scale," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pp. 423–438, 2013.

[25] A. Alexandrov, R. Bergmann, S. Ewen, J. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The stratosphere platform for big data analytics," *VLDB J.*, vol. 23, no. 6, pp. 939–964, 2014.

[26] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," *CoRR*, vol. abs/1506.08603, 2015.

[27] D. Borthakur, "Hdfs architecture guide," *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[28] A. Heise, A. Rheinländer, M. Leich, U. Leser, and F. Naumann, "Meteor/sopremo: an extensible query language and operator model," in *Workshop on End-to-end Management of Big Data, Istanbul, Turkey*, 2012.

[29] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/pacts: a programming model and execution framework for web-scale analytical processing," in *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pp. 119–130, 2010.

[30] A. Alexandrov, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, and D. Warneke, "Mapreduce and PACT - comparing data parallel programming models," in *Datenbanksysteme für Business, Technologie und Web (BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 2.-4.3.2011 in Kaiserslautern, Germany*, pp. 25–44, 2011.

[31] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, Ś 13, Farmington, PA, USA, November 3-6, 2013*, pp. 439–455, 2013.

[32] G. L. Sanders and S. Shin, "Denormalization effects on performance of RDBMS," in *34th Annual Hawaii International Conference on System Sciences (HICSS-34), January 3-6, 2001, Maui, Hawaii, USA*, 2001.

[33] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *PVLDB*, vol. 8, no. 12, pp. 1792–1803, 2015.

[34] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "Flumejava: easy, efficient data-parallel pipelines," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pp. 363–375, 2010.

[35] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: Fault-tolerant stream processing at internet scale," *PVLDB*, vol. 6, no. 11, pp. 1033–1044, 2013.

[36] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 2008.

[37] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pp. 261–264, 2012.

[38] S. Fernandes and J. Bernardino, "What is bigquery?," in *Proceedings of the 19th International Database Engineering & Applications Symposium, Yokohama, Japan, July 13-15, 2015*, pp. 202–203, 2015.

[39] M. Cohen, K. Hurley, and P. Newson, *Google Compute Engine*. " O'Reilly Media, Inc.", 2014.

Appendix A

Source Code

This chapter shows the location of each project that is used to test some functionalities of each framework. It provides a short description about the purpose of each project. All projects use Apache Hadoop and Kafka. Apache Hadoop is used to store checkpoints. Apache Kafka is used to read tweets and write results in a durable storage.

- **Apache Spark**
  - This project contains different test cases which are made on Apache Spark. It contains a class to test the normal word count program. Also, it contains another class to test the behavior of Spark streaming engine when having tweets from a specific number of users. This class is used to detect how Spark could manage session windowing.
  - **URL**: https://github.com/fsalem/Spark_Kafka_demo
- **Spark Writer to Kafka**
  - This project is Apache Kafka connector. It creates a poll of Kafka producers to write the content of Spark RDDs in a specific topic. It is able to manage the callbacks of Kafka producers. In addition, it contains listener to close all producers in case of terminating a Spark job.
  - **URL**: https://github.com/fsalem/Spark-Kafka-Writer
- **Apache Flink**:
  - This project contains the same test cases as in Apache Spark project. In addition, it contains another class to test different triggers such as *EventTimeTrigger*, *ContinuousEventTimeTrigger*, and the default trigger, which is *ProcessingTimeTrigger*
  - **URL**: https://github.com/fsalem/Flink-Kafka-demo
- **Apache Beam with Spark Runner**:
  - This project implements the same pipelines that are used in Apache Spark project to test the behavior of Spark runner.

  - **URL**: https://github.com/fsalem/Spark-Dataflow-demo

- **Apache Beam with Flink Runner**:
  - This project implements the same pipelines that are used in Apache Flink project to test the behavior of Flink runner. Additionally, it examines how Flink runner deals with session windowing.
  - **URL**: https://github.com/fsalem/Flink-Dataflow-demo